Informatik I

Di, Mi 8:15 - 10:00

Prof. Dr. Michael Hanus
mh@informatik.uni-kiel.de
880-7271
Raum 706

http://www.informatik.uni-kiel.de/~mh/lehre/info1-05.html

Übungen: Montag, 14:15-15:45, WSP3 Raum 1

Abgabe der Übungsblätter: Mittwochs per Mail / Briefkasten in Haus I

Erste Übungsstunde: 24.10.05

Literatur:

H. Abelson, G.J. Sussmann
„Struktur und Interpretation von Computerprogrammen"
Springer-Verlag, 2001
ISBN 3-540-42342-7
44,95€

M. Felleisen, R.B. Findler, M. Flatt, S. Krishnamurthi
"How to design programs"
MIT 2001
ISBN 0-262-06218-6
67$
http://www.htdp.org

18.10.2005

0. Grundbegriffe

- Programmieren: Formulierung eines Algorithmus in einer Programmiersprache
- Algorithmus: Beschreibung einer Vorgehensweise, wie man zu jedem aus einer Klasse gleichartiger Probleme eine Lösung findet
- Parameter: legt konkretes Problem der Klasse fest
- Korrektheit: a) partielle ~: falls Algorithmus anhält, dann ist Ergebnis eine Lösung
  b) totale ~: Algorithmus hält immer an und ist partiell korrekt

Eigenschaften von Algorithmen:
- Endlichkeit der Beschreibung
- Effektivität der Schritte
- Determiniertheit (evtl. verzichtbar)

Für manche mathematisch formulierbare Funktionen gibt es keine Algorithmen.

Programmiersprache (Hilfsmittel zur Formulierung von Algorithmen)
- Aspekte: Syntax: Was sind zulässige Zeichenfolgen
  Semantik: Was bedeuten die Zeichenfolgen
  Pragmatik: Wie wendet man die Sprache an
- Anforderungen: - Universalität
  - automatisch analysierbar

Programmiersprache: Scheme + Erweiterung „DrScheme“
http://www.schemers.org
http://www-ps.informatik.uni-kiel.de/~sebf/ws0506/info1

19.10.2005

1. Abstraktion mit Prozeduren

1.1 Programmelemente

Ausdrücke

- Zahlen: Wert ist die Zahl selbst
```
>42
42
```
- Kombination: Verknüpfung mit elementaren Prozeduren (Funktionen)
```
>(+ 10 31)
41
```
- Präfix-Notation: (<Operator> <Operand> <Operand>)

  Vorteile:
  - beliebig viele Argumente / Operanden (+ 1 2 3 4)
  - beliebige geschachtelte Ausdrücke (+ (+ 3 5) (- 8 6))

  Konvention zur Lesbarkeit: komplexe Ausdrücke mit vertikaler Ausrichtung:
```
(+ (* 3
      (+ (* 2 4)
         (+ 5 3)))
      (+ (-10 7)
         6))
```

2. Abstraktion durch Benennung von Objekten

Namensgebung: identifiziere Objekt durch Namen
auch: Name (Identifikator) benennt Variable, deren Wert ein Objekt ist
(define groesse 2)
(define qu-groesse (* groesse groesse))
⇒ abstrahiere Ausdrücke zu einem Namen
⇒ Scheme benötigt Speicher für Namenszuordnung
„Umgebung“: Zuordnung von Namen zu Objekten

3. Auswertung von Kombinationen

1. Alle Teile auswerten (bis auf primitive Teile): keine feste Reihenfolge
2. Wende Prozedur (linkeste Argument) auf restliche Elemente an
   - rekursiver Prozess, aber wohldefiniert (da Teile immer kleiner werden)

Bsp. (* (+ 2 (* 4 6)) (+ 3 5 7))
```
           390
    *   26          15
      + 2  24    + 3 5 7
          * 4 6
```

- Auswertung innen nach außen
- Auswertung parallel auf der gleichen Ebene (theoretisch)

4. Sonderform

z.B. (define x 3)

1. werte das letzte Element aus
2. binde das 2. Element an Wert des 3. Elements)

Namen der Sonderformen (z.B. define) := Syntax
Scheme: wenig Sonderformen => einfache Syntax

Bsp. Quadratfläche berechnen
Kantenlänge:
2 (* 2 2)
4 (* 4 4)
x (* x x)   x := Parameter, (* x x) := Rechenvorschrift

(define (quadrat x) (* x x))
quadrat := Prozedurname, x := Parameter, (* x x) := Rechenvorschrift

>(quadrat 4)
16

allgemein:
(define (<name> <formale Parameter>) <Rumpf>)
-> Ausdruck auch formale Parameter

- benutzbar wie elementare Prozeduren
- Bausteine für komplexere Prozeduren

<u>Substitutionsmodell:</u> Auswertung von Prozeduranwendungen
werte Rumpf aus, nachdem jeder formale Parameter durch entsprechendes Argument ersetzt wurde

Substitutionsmodell versagt bei veränderbaren Strukturen (später Umgebungsmodell)

25.10.2005

bisher: informelle Semantik
notwendig: für Programmierer + Implementierer: exakte Begriffe -> insbes. „Auswertung“
Berechnung: behandelt Terme / Ausdrücke (-> Zahlen, Funktionssymbole, Variablen)

Def.:
Signatur Σ: Menge von Paaren der Form f/n, wobei f ein Name und n eine ganze Zahl ist, die die Anzahl der Operanden von f angibt, mit: $f/n,\ \ f/m\ \in \Sigma$ impliziert m=n
Sprechweise: $f/n \in \Sigma$ und n>0 für n-stelliges Funktionssymbol
$c/0 \in \Sigma$ c Konstante
Anmerkung:
- manchmal Eindeutigkeit nicht gefordert: „überladene“ Bezeichner
- häufig: Funktionen mit Typangaben (hier nicht, wegen Scheme)

Bsp: $\Sigma = \{+/2,\ -/2,\ */2,\ quadrat/1,\ quadratsumme/2\}$

Annahme: Zahlen sind (implizit) Konstanten, d.h. $n/0 \in \Sigma$ für alle Zahlen n

Def.:
Sei Σ Signatur, X Menge von Variablen (Namen verschieden von Symbolen in Σ sein).
$T_\Sigma(X)$ Terme über Σ und X wie folgt:
1. für alle Konstanten $c/0 \in \Sigma$ gilt: $c \in T_\Sigma(X)$
2. für alle Variablen $x \in X$ gilt: $x \in T_\Sigma(X)$
3. Ist $f/n \in \Sigma$ (n>0) und $t_1,...,t_n \in T_\Sigma(X)$, dann $(f\ t_1,...,t_n) \in T_\Sigma(X)$

Beachte:
- rekursive / induktive Definition: typisch für Objekte in der Informatik
- formaler: $T_\Sigma(X)$ kleinste Menge, die 1+2+3 erfüllt

Bsp:
Σ wie oben, $X = \{x, y, z\}$ dann: $(+\ 1\ 2) \in T_\Sigma(X)$
$(*\ x\ (+\ y\ z)) \in T_\Sigma(X),\ \ (quadratsumme\ (*\ x\ 2)) \in T_\Sigma(X)$

Def.:
Programm ist Tripel (Σ, X, R) mit:
- Σ ist Signatur
- X ist Variablenmenge disjunkt zu Σ
- R ist Menge von Regeln der Form $(f\ \ x_1,...,x_n) \to r$ mit $f/n \in \Sigma$, $x_1,...,x_n \in$ paarweise verschieden, $r \in T_\Sigma(\{x_1,...,x_n\})$

Anmerkungen:
- häufig: nur eine Regel pro Funktion
- rechte Regelseiten: nur Signatursymbole und linke Parameter
- manchmal: linke Regelseite: Term (-> Termersatzsysteme)
- Scheme: Definition $(define\ (f\ \ x_1,...,x_n)\ r)$ entspricht obiger Regel

Def.:

Substitution σ ist Funktion $\sigma : X \to T_\Sigma(X)$, die jeder Variablen einen Term zuordnet
Anwendung einer Substitution σ auf Term $t \in T_\Sigma(X)$, geschrieben $t\sigma$, ist wie folgt definiert (induktiv):

- falls $t \in X$, dann $t\sigma = \sigma(t)$
- falls $t/0 \in \Sigma$, dann $t\sigma = t$
- falls $t = (f\ t_1,...,t_n)$, dann $t\sigma = (f\ t_1\sigma,...,t_n\sigma)$

Bsp:
σ Substitution mit $\sigma(x) = 2\ \ \sigma(y) = (+\ z\ 1),\ \sigma(z) = z$
dann $(*\ x\ (+\ y\ z))\sigma = (*\ 2\ (+\ (+\ z\ 1)\ z))$

Def.:

- Position: Identifikation einer Stelle im Term. Pos(t): Menge aller Positionen im Term t
  üblich: Positionen ≈ Folgen natürlicher Zahlen.
  leere Folge <>: Position der „Wurzel" (Einstiegspunkt)
  nichtleere Folge < $p_1,..., p_n$ >: Im Argument p_1 die Position < $p_2,..., p_n$ >
  Bsp: t=(* x (+ y z))
  $pos(t) = \{<>,<1>,<2>,<2,1>,<2,2>\}$
- Teilterm: $p \in pos(t)$, dann: $t|_p$ Teilterm von t an Position p:
  - p=<>: $t|_{<>} = t$
  - p=< $p_1,..., p_n$ > und $t = (f\ \ t_1,...,t_m)$. $t|_{<p_1,...,p_n>} = t_{p_1}\ |_{<p_1,...,p_n>}$

  t wie oben: $t|_{<1>} = x|_{<>} = x$
  $t|_{<2,2>} = (+\ y\ z)|_{<2>} = z|_{<>} = z$
- Ersetzung: $p \in pos(t)$ und t'Term $t[t']_p$ Ersetzung von $t|_p$ durch t'
  - p=<> $t[t']_{<>} = t'$
  - p=< $p_1,..., p_n$ > und $t = (f\ \ t_1,...,t_m)$:
    $t[t']_{<p_1,...,p_n>} = '(f\ \ {}_{t_1,...,t_{p_1-1}}[t']_{<p_1,...,p_n>} t_{p1+1)} ....t_m)$

  t wie oben, t'=(+ z y). $t[t']_{<2>} = (*\ x\ (+\ z\ \ y))$

Def.
Sei P= (Σ, X, R) Programm

- Berechnungsschritt $t_1 \Rightarrow t_2$ (formal => Relation auf Termen) falls
  $p \in pos(t_1),\ l \to r \in R$ und σ Substitution mit:

1. $\sigma(l) = t_1|_p$ (σ ersetzt formale Parameter durch aktuelle Terme)
2. $t_2 = t_1[\sigma(r)]_p$ (Ersetzung mit Parametersubstitution)

- Berechnung $t_1 \Rightarrow^* t_2$ ist eine (evtl. leere) Folge von Berechnungsschritten. d.h. es existieren n und $s_1,...s_n \in T_\Sigma(X)$ mit $t_1 = s_1,\ t_2 = s_n$ und $s_i \Rightarrow s_{i+1}\quad (i = 1,\ ...,\ n-1)$

Bsp:
(quadrat x) -> (* x x)
(quadrat (+ 1 2) => (* (+ 1 2) (+ 1 2)) (p=<>, $\sigma(x) = (+\ 1\ 2)$)
(* 3 (quadrat 4)) => (* 3 (* 4 4)) (p=<2>, $\sigma(x) = 4$)

Auswertung vordefinierter Funktionen:
Konzeptionell definiert durch unendliche Menge von Regeln:

```
(* 0 0) -> 0              (* 2 3) -> 6
(+ 0 1)-> 1               …
Damit. (quadrat (+ 1 2))     => (* (+ 1 2) (+ 1 2))
                             => (* 3 (+ 1 2))
                             => (* 3 3)
                             => 9
auch: (quadrat (+ 1 2))      => (quadrat 3)
                             => (* 3 3)
                             => 9
```

Def.:
Term t heißt Normalform, falls es kein t’ gibt mit t=>t’
Normalformen ≈ Endergebnisse ≈ Werte

(quadrat (+ 1 2))
zuerst -> Berechnung applikativer Ordnung
zuerst -> Normalordnung

Auswertung in applikatver Ordnung:
Informell: werte Argumente vor Anwendung einer Prozedur aus
Formal: Falls $t_1 \Rightarrow t_2 \;\; mit \;\; t_2 = t_1[\sigma(r)]_p, \; dann \; ist \; t_1|_p = (f \;\; s_1...s_n)$ mit jedes s_1 in Normalform

Auswertung in Normalordnung:
Informell: ersetze äußerste Funktionsaufrufe (falls das möglich ist) zuerst
Formal: Falls $t_1 \Rightarrow t_2 \;\; mit \;\; t_2 = t_1[\sigma(r)]_p$, dann ist $t_1|_p$ äußerster Teilterm, an dem ein Schritt möglich ist (d.h. falls $p = < p_1...p_n >$, dann ist Schritt an $< p_1...p_i >$ mit $i < n$ nicht möglich)
Bsp:
(f 5)
=> (quadratsumme (+ 5 1) (* 5 2))
=> (+ (quadrat (+ 5 1)) (quadrat (* 5 2)))
$=>^2$ (+ (* (+ 5 1) (+ 5 1)) (* (* 5 2) (* 5 2)))
=> (+ (* 6 6) (* 10 10))
=> (+ 36 100)
$=>^7$ 136

Beobachtung: Doppelauswertung von (+ 5 1) und (* 5 2)
(Verbesserung: lazy Auswertung)
Hier: applikative + Normalordnung liefern identisches Ergebnis
jedoch: Normalordnung kann Ergebnis liefern, bei der applikative Ordnung nicht terminiert
- ⇨ Normalordnung liefert Normalform, falls diese existiert
- ⇨ (define (f x) 0)
- ⇨ (define (g y) (g (+ y 1)))

Normalordnung: (f (g 0)) => 0
applik. Ordnung: (f (g 0)) => (f (g (+ 0 1))) => (f (g 1)) => (f (g (+ 1 1))) …

Aber: Scheme benutzt applikative Ordnung <- im allg. effizienter

Bedingte Ausdrücke:

Absolutbetrag: $abs(x) = \begin{cases} x \;\; falls \;\; x > 0 \\ 0 \;\; falls \;\; x = 0 \\ -x \;\; falls \;\; x < 0 \end{cases}$

Sonderform zur Fallunterscheidung: cond.

$(cond \;\; (< p_1 > < a_1 >)$
$(< p_2 > < a_2 >)$ <- Klausel
$(< p_n > < a_n >))$

Bedingung, Prädikat: Ausdrücke mit Ergebnis: true / false (Bsp.: < <= > =)

Bedeutung:

1. werte $< p_1 >$ aus: falls Wert=true, dann Ergebnis: Wert von $< a_1 >$
2. werte $< p_2 >$ aus: falls Wert=true, dann Ergebnis: Wert von $< a_2 >$
3. werte $< p_n >$ aus: falls Wert=true, dann Ergebnis: Wert von $< a_n >$

   Sonst. Ergebnis: undefiniert (= Fehler)

Bsp:

```
(define (abs x)
        (cond  ((> x 0) x)
               ((= x 0) 0)
               ((< x 0) (-x)))))
```

Spezialfall: $< p_n > = else \ (< p_n >= true)$

```
=> define (abs x)
        (cond ((> x 0) x)
               (else (- x)))
```

Weitere Sonderform:

$(if \ < p > < a_1 > < a_2 >)$

entspricht $\begin{array}{l}(cond \quad (< p > < a_1 >) \\ \qquad\qquad (else < a_2 >))\end{array}$

Bsp: (define (abs x) (if (> x 0) x (-x)))

Zusammengesetzte Prädikate

$(and < p_1 > ... < p_n >)$: true, falls alle $< p_1 >$ den Wert true haben

$(or < p_1 > ... < p_n >)$: false, falls alle $< p_1 >$ den Wert false haben

$(not < p >)$:true, falls Wert $< p >$ false, sonst false

Bsp.:

Größer-gleich:

(define (>= x y (or (> x y) (= x y)))

Quadratwurzelberechnung nach Newton

Math. Wurzelfunktion $\sqrt{x} = y \ falls \ y \geq 0 \ und \ y^2 = x$

präzise Funktion, aber nicht effektiv (Unterschied Mathematik <-> Informatik)

Berechnungsverfahren: Newtonsches Iterationsverfahren (Approximationsverfahren)

Gegeben: Schätzung y

bessere Schätzung: Mittelwert von y und x/y

Bsp.: x=2

Schätzung:	Mittelwert:
1	(1+2)/2=1,5
1,5	(1,5+1,2233)/2=1,4167
1,4167	(1,4167+1,4118)=1,4142

Iteration:

```
(define (wurzel-iter y x)
        (if (gut-genug? y x)
        y
        (wurzel-iter (verbessern y x) x)))
(define (verbessern y x) (mittelwert y (/ x y)))
(define (mittelwert  x y) (/ (+ x y) 2))
```

Abbruchkriterum (vereinfacht!): $|y^2 - x| < 0,001$

```
(define (gut-genug? y x) (< (abs (- (quadrat y) x)) 0.001))
```

Gesamtlösung:

```
(define (wurzel x) (wurzel-iter 1 x))
```

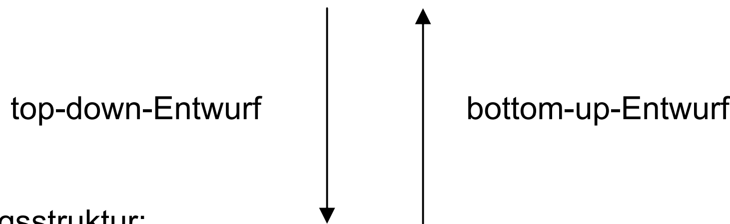
- Scheme rechnet beliebig genau (ungenau nur bei ungenauen Konstanten: (wurzel #i9))
- numerische Anwendungen möglich
- keine iterativen Konstrukte (schleifen) notwendig
- Prozeduraufrufe: Schleifen = Rekursion
- Effizienz: kein Problem (später mehr)

1.11.2005

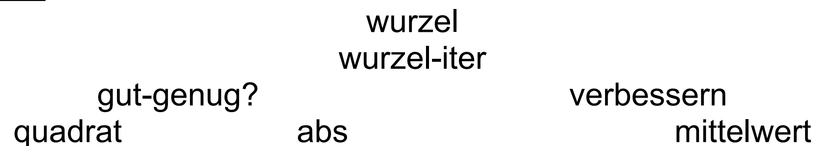
Vorgehensweisen:

- Problem in Teilprobleme zerlegt
- Teilprobleme ≙ Prozeduren
- Prozeduren: wichtig: erfüllen bestimmte Aufgabe
  unwichtig: wie erfüllen sie die Aufgabe?

=> prozedurale Abstraktion „black box“



Zerlegungsstruktur:



Namen:

- Prozedurnamen wichtig (Abstraktion, gut gewählt!)
- Namen lokaler Variablen nur intern relevant
  (define (quadrat x) (* x x)) ->
  (define (quadrat y) (* y y)) -> äquivalent
  x / y := lokaler Name
- wichtig, um Prozeduren unabhängig zu entwickeln
- formale Parameter: gebundene Variablen
- Gegensatz: freie Variablen (z.B. abs, quadrat, …)

1.2 Prozeduren und Prozesse

Prozeduren: Berechnungsmuster
Prozesse: dynamischer Berechnungsablauf, gesteuert durch Prozeduren

Frage: Ressourcenverbrauch (Speicher, Zeit) von Prozessen

Lineare Rekursion und Iteration

Bsp: Fakultät: $n! = n(n-1)(n-2)...*2*1$
$= n(n-1)! \qquad (falls\ n > 1)$

=> (define (fak n) (if (= n 1) 1 (* n (fak (- n 1)))))

Berechnungsprozess für (fak 3) (nach Substitutionsmodell, ohne if-Auswertung)

```
(fak 3)
(* 3 (fak (- 3 1)))
(* 3 (fak 2))
(* 3 (* 2 (fak 1)))          Länge des Terms: Speicherverbrauch
(* 3 (* 2 1))
(* 3 2)
6
```

Alternativ: Multipliziere 1*2, das Ergebnis mit 3, das Ergebnis mit 4, …, mit n
Parameter in jedem Schritt:

- Zähler (1 bis n)
- Produkt (1 bis n!)
- n (für Abbruch)

=>
```
(define (fak n) (fak-iter 1 1 n))
(define (fak-iter p z max) (if (> z max) p (fak-iter (* p z) (+ z 1) max)))
```

Berechungsprozess:
(fak 3)
(fak-iter 1 1 3)
(fak-iter 1 2 3) konstanter Speicherverbrauch
(fak-iter 2 3 3)
(fak-iter 6 4 3)
6

Beobachtung:

- beide Berechnungen haben gleiche Länge (linearer Prozess)
- 1. Version: Multiplikation „verzögert", zuerst rekursiver Aufruf => linearer rekursiver Prozess (Speicher: linear in der Eingabegröße)
- 2. Version: schrittweise Multiplikation, p, z, max beschreiben Zwischenzustand jedes Schrittes
  => Zustandsvariablen
  => linearer iterativer Prozess (Speicherbedarf konstant)

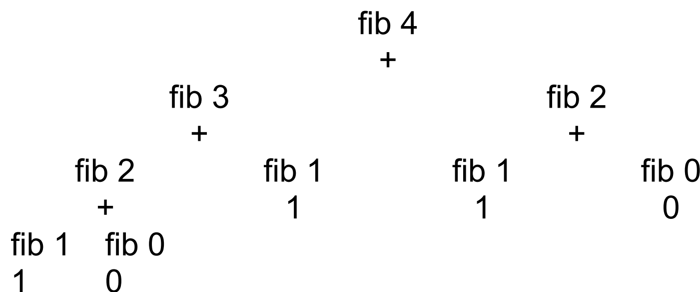
Rekursive Prozeduren $\neq$ Rekursive Prozesse

Baumrekursion
Bsp: Fibonacci-Zahlen
Jede Zahl ist die Summe der beiden vorgehenden (Start: 0 1 1 2 3 5 8 13 21 34 …)

$$Fib(n) = \begin{Bmatrix} 0 \;\; falls \;\; n = 0 \\ 1 \;\; falls \;\; n = 1 \\ Fib(n-1) + Fib(n-2) \end{Bmatrix}, \text{ sonst}$$

=> (define (fib n) (cond ((= n 0) 0) ((= n 1) 1) (else (+ (fib (- n 2)) (fib (- n 2))))))

Berechnungsprozess:



baumrekursiver Prozess

Redundanz: z.B. (fib 2) wird 2-mal berechnet
letzendlich: nur Einsen addiert
=> expotentielle Laufzeit wegen $Fib(n) = \left[\frac{\phi^n}{\sqrt{5}} \right]$ mit $\phi = \frac{1+\sqrt{5}}{2} \approx 1{,}6180$

Iterativer Prozess: Aufsummieren der Zahlen
Parameter: letzte Zahlen: a, b (vorletzte Zahl)

Schritt: a <- a+b
b <- a gleichzeitig

```
(define (fib n) (fib-iter 1 0 n))
(define (fib-iter a b z) (if (= z 0) (fib-iter (+ a b) a (- z 1))))
```

⇨ linearer iterativer Prozess (lineare Laufzeit, konstanter Speicher)

<u>Größenordnungen</u>

- wichtig zum Vergleich des Ressourcen-Verbrauchs
- gemessen in Abhängigkeit von einem Parameter n (Genauigkeit bei der Wurzelberechnung, n. Zahl, Matrixgröße)
- R(n): Betrag an Ressourcen des Prozesses zur Berechnung bezüglich n
- exakt schwer zu messen (abhängig von Implementierung)
- => R(n) hat <u>Größenordnung</u> $O(f(n))$ falls es Konstanten c und n_0 (unabhängig von n!) gibt mit $R(n) \leq c * f(n)$ für alle $n \geq n_0$

Bsp: Lin. reh. Fakultätsprozess:

- Zeit: O(n)
- Speicher: O(n)

Lin. it. Fakultätsprozess:

- Zeit: O(n)
- Speicher: O(1)

Fibonacci:

- Zeit: $O(\phi^n)$
- Speicher: O(1)
- Speicher: O(n)

- Grobes Maß für Effizienz: Prozesse mit expon. Laufzeit $O(c^n)$ in der Praxis unbrauchbar

- Allerdings: viele interessante Probleme, haben expon. Laufzeit!

Potenzrechnung:
Eingabe: b, n
Ausgabe: b^n
Unmittelbar:
- Rekursiv:
  $b^n = b*b*b*...*b\ [n\ Mal] = b*b^{n-1}, n > 1$
  $b^0 = 1$
  (define (potenz b n) (if (= n 0) 1 (* b (potenz b (- n 1)))))
  => linearer rekursiver Prozess
- Iterativ:
  (define (potenz b n) (pot-iter n 1 b))
  (define (pot-iter z p b) (if (= z 0) p (pot-iter (- z 1) (* p b) b)))
  Zeitverbrauch: O(n) -> linear abhängig von n
  Speicherverbrauch: O(1) -> konstant

Schnellere Berechnung: Quadratbildung
statt: $b^8 = b*b*b*b*b*b*b*b$
$b^2 = b*b$
$b^4 = b^2*b^2$
$b^8 = b^4*b^4$

=> Allgemein: $b^n = \left(b^{\frac{n}{2}}\right)^2$ falls n gerade
$b^n = b*b^{n-1}$ falls n ungerade

```
(define (schnell-pot b n)
                    (cond ((= n 0) 1)
                          ((gerade? n) (quadrat (schnell-pot b (/ n 2))))
                          (else (schnellpot b (- n 1))))))
(define (gerade? n) (= (remainder n 2) 0))
```

remainder: Rest ganzzahliger Division = modulo

Laufzeit: Unterschied b^n und b^{2n}: 1 Aufruf
=> Anzahl der Aufrufe / Multiplikationen ~ Logarithmus (zur Basis 2) von n
=> Zeit: O(log n) gut bei großen Zahlen, z.B. $b^{1000000}$ ≈ 20 Multiplikationen

Größter gemeinsamer Teiler

ggt(a,b) größte ganze Zahl, die sowohl a als auch b teilt.
ggt(16,24)=8
Wie berechnen?
1. Probiere alle Zahlen als Teiler
2. Zerlege Zahlen in Primfaktoren, multipliziere gemeinsame
3. Euklidischer Algorithmus: ggt(a,b)=ggt(b, Rest von a/b)
   ggt(206,40)=ggt(40,6)=ggt(6,4)=ggt(4,2)=ggt(2,0)=2
   Feststellung: Reduktion führt immer zu ggt(x,0)

```
(define (ggt a b)
    (if (= b 0)
        a
        (ggt b (renainder a b))))
```
iterativer Prozess, Laufzeit?

Satz von Lané (1845): Euklidscher Alg. benötigt k Schritte => $\min(a,b) \geq Fib(k)$
Laufzeit: Sei n=min(a,b)
=> Algorithmus macht k Schritte mit $n \geq Fib(k) \approx \phi^k$
=> Schrittzahl kleiner als $\log_\phi n$
=> Laufzeit O(log n)

Primzahltest

Klassische Methode: Suche nach Teilern: probiere alle von 2 bis $\sqrt{n}$
```
(define (kleinster-teiler n) (finde-teiler 2 n))
(define (finde-teiler test n)
    (cond ((> (quadrat test) n) n)
        ((teilt? test n) test)
        (else (finde-teiler (+ test 1) n))))
(define (teilt? a b) (= (remainder b a) 0))
=> (define (primzahl? n) (= (kleinster-teiler n) n))
```

Iterativer Prozess, Laufzeit: O($\sqrt{n}$)
Für große Zahlen (> 100 Stellen): zu langsam, Wunsch: O(log n)
Ergebnis aus Zahlentheorie: Kleiner Fermatscher Satz: n Primzahl, 0<a<n: $a^n \bmod n = a$
Falls n keine Primzahl ist, dann gibt es (wahrscheinlich) viele Zahlen 0<a<n: $a^n \bmod n = a$ gilt nicht.
Für zufälliges a: teste $a^n \bmod n = a$. Mehr Tests => höhere Primzahlwahrscheinlichkeit

Potenzbildung: modulo n: wie schnell-pot
```
(define (potmod b e n)
    (cond ((= e 0) 1)
        ((gerade? e) (remainder (quadrat (podmod b (/ e 2) n)) n))
        (else (remainder (* b (potmod b (- e 1) n)) n))))
```
Fermat-Test: Wähle Zufallszahl a mit $2 \leq a \leq n-1$, teste $a^n \bmod n = a$
Zufallszahl: (random x): liefert ganze Zufallszahl $0 \leq z < x$
```
(define (fermat-test n)
    (test-mit-zufall n (+ 2 (random (- n 2)))))
(define (test-mit-zufall n a) (= potmod a n n) a))
```
x-malige Ausführung:
```
(define (fast-prime? n x)
    (cond ((= x 0) true)
        ((fermat-test n) (fast-prime? n (- x 1)))
        (else false)))
```

=> Probalistische Methode: Ergebnis false: sicher keine Primzahl
Ergebnis true: könnte Primzahl sein

Argument für prob. Algorithmen: auch „korrekte“ Algorithmen können fehlschlagen (z.B. Hardwarefehler)

8.11.2005

2. Abstraktion mit Daten

Abstraktion mit
- Prozeduren: interne Berechnung: unsichtbar
- Daten: interne Darstellung: unsichtbar (wichtig: Anlegen von Daten, Operationen, z.B. Zahlen)

Abstrakte Daten:
- Anwender: kennt nicht interne Struktur, benutzt nur Operationen
- Implementierer wählt interne Struktur, implementiert Operationen

Vorteile:
- interne Implementierung leicht austauschbar (Effizienz!)
- mehr Modularität
- bessere Verständlichkeit der Programme

Schnittstelle:
- Selektoren: Extraktion von Teilformationen
- Konstruktoren: Konstruktion komplexer Daten
- Operatoren: Verknüpfung der Daten

Beispiel: Rationale Zahlen

z: Zähler ganzzahlig
n: Nenner ganzzahlig

Konstruktor: (konstr-rat z n)
Selektoren: (zaehler r) liefert Zähler einer rationalen Zahl r
(nenner r) liefert Nenner einer rationalen Zahl r
Operatoren: (+rat r1 r2), (*rat r1 r2), …
können mit Konstruktor / Selektoren implementiert werden

$$\frac{z_1}{n_1} + \frac{z_2}{n_2} = \frac{z_1 n_2 + z_2 n_1}{n_1 n_2}$$

=>

```
(define (+rat r1 r2) (konstr-rat (+  (* (zaehler r1) (nenner r2))
                                     (* (zaehler r2) (nenner r1))
                                  (* (nenner r1) (nenner r2))))
```

$$\frac{z_1}{n_1} * \frac{z_2}{n_2} = \frac{z_1 z_2}{n_1 n_2}$$

=>

```
(define (*rat r1 r2) (konstr-rat (* (zaehler r1) (zaehler r2))
                                 (* (nenner r1) (nenner r2))))
```

Strukturdefinition:
Struktur ≈ Kombination von Daten zu einer Einheit
Sonderform in DrScheme (kein Standard)

(define-struct sname (k_1 ... k_n))
sname: Strukturname, k_1 ... k_n : Komponentennamen
z.B. (define-struct rat (zaehler nenner))
=> Operationen werden definiert

1. (make-sname x_1 ... x_n): Erzeuge Struktur
2. (sname-k_i s): liefert Komponente k_i der Struktur s
3. (sname? s): true falls s eine sname-Struktur ist

z.B.
>(make-rat 1 3)
(make-rat 1 3)
>(rat-nenner (makerat 1 3))
3

Intuition: Struktur ≈ Box mit Schubladen, die Namen haben:
(make-rat 1 3) → zaehler → 1, nenner → 3

„Kasten-Zeiger-Darstellung“

Wichtig sind nur Eigenschaften / Gesetze.
Es gilt immer: Sei s Wert von (make-sname x_1 ... x_n)
Dann gilt: (sname-k_1 s)= x_1
(sname-k_n s)= x_n
(sname? s)=true

Kurzschreibweise:
(sname-k_i (make sname x_1 ... x_n))= x_i

Konkret: rat-Struktur:

1. Gesetz: (rat-zaehler (make-rat z h))=z
2. Gesetz: (rat-nenner z n))=n

z.B.
>(define r (make-rat 2 5))
>(+ (rat-zaehler r) 3)
(makerat 2 5)
=> (+ 2 3) => 5

Erweitertes Substitutionsmodell von obigen Gesetzen

Implementierung rationaler Zahlen:
(define (konst-rat z n) (make-rat z n))
(define (zaehler r) (rat-zaehler r))
(define (nenner r) (rat-nenner r))
>(define ein-drittel (konst-rat 1 3))
>(+rat eindrittel eindrittel)

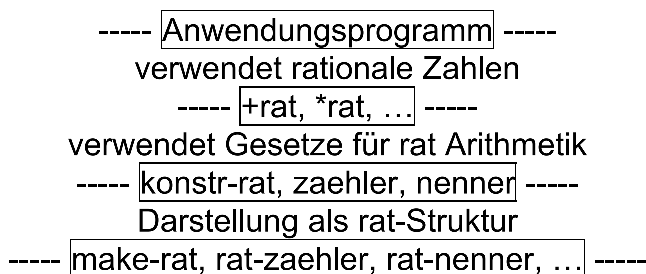
(make-rat 6 9)   ← nicht gekürzt, Abhilfe: bei Konstruktion ggt-Division

(define (konstr-rat z n) (make-rat (/ z (ggt z n)) (/ n (ggt z n))))
>(+rat eindrittel eindrittel)
(make-rat 2 3)

Vorteil der Abstraktion von konstr-rat ändern, nicht jedoch +rat, *rat

Konstruktionsmethode: Schichtenentwurf von Programmen
Unterste Schicht: elementare Objekte, Objekte der Programmiersprache
Oberste Schicht: Anwendungsobjekte

wichtig: kein „Durchgreifen" durch Schichten, nur Schnittstellen benutzen
-> Abstraktionsbarrieren



Vorteil:
- Implementierung einzelner Schichten austauschbar
- Beherrschung der Entwurfskomplexität
- Verifikation einzelner Schichten

Abstrakte Datentypen
Wichtig: Schnittstelle (z.B. konstr-rat, zaehler, nenner) hat Gesetze, die jede korrekte Implementierung erfüllen muss.
Z.B. (zaehler (konstr-rat z n)) / (nenner (konstr-rat z n)) = z / n
=> Datentypen => Selektoren, Konstruktoren, Operatoren + Geseze
abstrakt: keine konkrete Implementierung festgelegt (keine Kürzung notwendig)

Definition:
Ein abstrakter Datentyp (ADT) (Σ,X,E) besteht aus
- Signatur Σ (enthält alle Dateitypoperationen)
- Variablenmenge X disunkt zu Σ
- Menge E von Gleichungen der Form $t_1 = t_2$ mit $t_1, t_2 \in T_\Sigma(X)$ (Terme)

Gleichungen	$\neq$	Regeln
müssen durch Implementierung erfüllt sein		Programmablauf, Anwendung links nach rechts

Definition:
Programm (Σ,X,R) ist Implementierung eines ADT (Σ,X,E), falls dieses alle Gleichungen in E erfüllt, d.h. falls $t_1 = t_2 \in E$ und $\sigma \quad X \to T_\Sigma(X)$ und es gilt
$\sigma(t_1) \Leftrightarrow s_1 \Leftrightarrow s_2 \Leftrightarrow ... \Leftrightarrow s_n \Leftrightarrow \sigma(t_n)$.
($s \Leftrightarrow s'$ falls $s \Rightarrow s'$ oder $s' \Rightarrow s$)

ADT: Signatur + Gesetze: $t_1 = t_2 \quad \forall \sigma : \sigma(t_1) \Leftrightarrow ... \Leftrightarrow \sigma(t_2)$ bzgl. Programm R

Beispiel: Implementierung rationaler Zahlen ist korrekt

$$\frac{\text{(zaehler (konstr-rat z n))}}{\text{(nenner (konstr-rat z n))}} \stackrel{!}{=} \frac{z}{n}$$

$$\Rightarrow^2 \frac{\text{(rat-zaehler (konstr-rat z n))}}{\text{(rat-nenner (konstr-rat z n))}} \stackrel{!}{=} \frac{z}{n}$$

$$\Rightarrow^2 \frac{\text{(rat-zaehler (make-rat z n))}}{\text{(rat-nenner (make-rat z n))}} \stackrel{!}{=} \frac{z}{n} \qquad \text{| 1. Gesetz für rat-Struktur}$$

$$\Rightarrow \frac{z}{\text{(rat-nenner (make-rat z n))}} \stackrel{!}{=} \frac{z}{n} \qquad \text{| 2. Gesetz für rat-Struktur}$$

$$\Rightarrow \frac{z}{n} \stackrel{!}{=} \frac{z}{n}$$

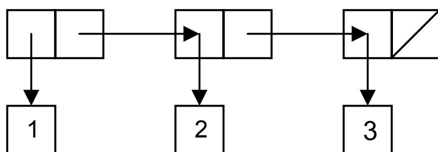
Analog: 2. Implementierung mit ggt ist korrekt

2.2 Hierarchische Datenstruktur

Häufig: Strukturen mit beliebig vielen Komponenten (z.B. Einkaufslisten,…)
Liste / Sequenz: geordnete (<- Reihenfolge) Menge (<- beliebig groß) von Objekten
Modellierung durch Fallunterscheidung

1. Fall: Liste ist leer: empty
2. Fall: Liste nicht leer: hat erstes Element und Restliste
   (cons <1.Element> <Restliste>)

Beispiel:
(cons 1 empty): Liste mit Element 1
(cons 1 (cons 2 empty)): Liste mit Elementen 1 und 2
in Kasten-Zeiger-Darstellung:



Zugriff aus Komponenten von cons: first und rest

Gesetze:
(first (cons e l))=e
(rest (cons e l))=l
Achtung: cons prüft zweites Argument auf Liste:
>(cons 1 2)
Error: 2 ist not a list

Spezielle Schreibweise für Listen:
(list <a_1>…<a_n>)
≡(cons <a_1> (cons <a_2> (…. (cons <a_n> empty))…

```
>(define 1-3 (list 1 2 3))
>1-3
(const 1 (cons 2 (cons 3 empty)))
>(first 1-3)
1
```

Operationen auf Listen

n-tes Element einer Liste (Nummerierung von 0 an)

```
>(n-tes 1 1-3)
2
```

n=0 => (first l)
n>0 => (n-1)-tes Element der Restliste

```
(define (n-tes n l)
        (if (= n 0)
        (first l)
        (n-tes (- n 1) (rest l))))
>(n 2 1-3)
3
```

Schema:

```
(define (listop …l…)
        (if (…)                  <- Bedingung für 1. Element
                (… (first l) …)
                (… (rest l)…)))
```

Anderes Schema:
Fallunterscheidung leer / nicht leer?
Hierzu: Prädikat: <u>empty?</u>: ist Argument eine leere Liste?
Gesetze: (empty? empty)=true
(empty? (cons e l))=false

Analog: <u>cons?</u>: ist Argument nicht-leere Liste?

=> Schema:

```
(define (listop … l …)
        (if (empty? l)
                (…)
                (… (first l) … (rest l) …)))
```

Anwendung: Länge einer Liste:

```
(define (laenge l)
        (if (empty? l)
        0
        (+ 1 (laenge (rest l)))))
```

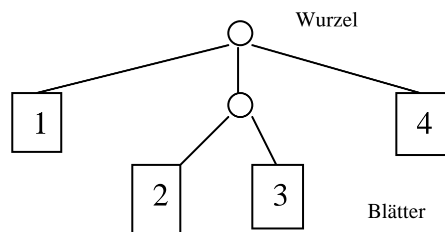
<(laenge 1-3)
3

Häufig: Analyse / Durchlaufen einer Liste + Konstruieren einer Ergebnisliste
Beispiel:
(append l1 l2): Liste mit Elementen aus l1 und l2 (gleiche Reihenfolge):
>(append 1-3 1-3) -> (list 1 2 3 1 2 3)

```
(define (append l1 l2)
        (if (empty? l1)
        l2
        (cons (first l1) (append (rest l1) l2))))
```

Baumstruktur

Liste: sequentielle Anordnung
häufig: hierarchische Anordnung =>
<u>Baum</u>: Datenstruktur mit einer Wurzel, <u>innere Knoten</u> mit <u>Sohnknoten</u> (Wurzeln von Teilbäumen) und <u>Blätter</u>



Modellierung durch Fallunterscheidung:
1. Fall: Baum ist Blatt, d.h. eine Zahl
2. Fall: Baum ist innerer Knoten, d.h. Liste der Söhne

Obiger Baum: (list 1 (list 2 3) 4)

zur Fallunterscheidung: (<u>number?</u> n): true falls n Zahl
Wann ist Baum ein Blatt?
(define (leaf? b (number? b))

Anwendung: Anzahl Blätter in einem Baum:

```
(define (blattanzahl b)
        (cond ((leaf? b) 1)
                ((empty? b) 0)                  Knoten ohne weitere Söhne
                (else (+ (blattanzahl (first b)) (blattanzahl (rest b))))))
```

15.11.2005

Quotieren:

Wunsch: nicht nur Zahlen verarbeiten, sondern auch Symbole
Beispiel: Liste mit Symbolen a, b, c: (list a b c)
Baum mit Namenssymbolen (list Norbert (list Werner Hans) Otto)
bisher nicht möglich: (list a b c) -> Error: undefined a…
Ursache: Scheme wertet alle Teilausdrücke aus
Lösung: Verhinderung der Auswertung für Symbole durch Quotieren (to quote = „zitieren“)
in Scheme: Apostroph vor Symbol

```
>(define (a 1))
>(define (b 2))
>(list a b)          -> (cons 1 (cons 2 empty))
>(list ’a ’b)        -> (cons ’a (cons ’b empty))
```

Unterschied:
Kiel ist eine schöne Stadt.
„Kiel“ ist der Name einer Stadt.
quotiert
Notwendig: Gleichheit von Symbolen: eq?
Beispiel:
Extrahiere Restliste einer Symbolliste, die mit bestimmtem Symbol beginnt
(nicht vorhanden: false)

```
(define (memq e l)
  (cond ((empty? e) false)
  ((eq? e (first l)) l)
  (else (memq e (rest l)))))

>(memq ’b (list ’a ’b ’e))          -> (cons ’b (cons ’e empty))
```

Beispiel: Symbolisches Differenzieren:

- nicht numerisch, sondern symbolisch: ax²+bx+c -> 2ax+b
- Funktionen mit Zahlenkonstanten, Variablen, Summen, Produkte
- Darstellung: - Stringdarstellung „ax+b“ -> (a * x + b)
  - Präfixdarstellung (á la Scheme): „ax+b“ -> (+ (* a x) b) ≈ Baumdarstellung
    günstiger: Prioritäten / Assoziativitäten trivial

Datentyp zur Funktionsdarstellung:
Konstruktoren: Zahlen, Variablen (Symbole): trivial

```
(define (konstr-summe a1 a2) (list ’+ a1 a2))
(define (konstr-produkt a1 a2) (list ’* a1 a2))
```

Prädikate: (wichtig, da Funktionsdarstellung Vereinigung ist)

- Konstante: (define (konstante? x) (number? x))
- Variable: (define (variable? x) (symbol? x))
- Summe: (define (summe? x) (if (cons? x) (eq? (first x) ’+) false))
- Produkt: (define (produkt? x) (and (cons? x) (eq? (first x) ’*)))
- Vergleich von Variablen:
  (define (gleiche-variable? v1 v2) (and (variable? v1) (variable? v2) (eq? v1 v2)))

Selektoren: Operanden von Summe+Prädikaten

```
(define (operand1 x) (n-tes 1 x))
(define (operand2 x) (n-tes 2 x))
```

Implementierung des Differenzierens: Umsetzung von Ableitungsregeln

$\frac{dc}{dx}=0$ (c Konstante oder Variable $\neq$ x)

$\frac{dx}{dx}=1$

$\frac{d(u+v)}{dx}=\frac{du}{dx}+\frac{dv}{dx}$ $\frac{d(uv)}{dx}=u\frac{dv}{dx}+v\frac{du}{dx}$

```
(define (ableitung a v)                a: Funktionsausdruck, v: Variable, z.B. 'x
  (cond ((konstante? a) 0)
    ((variable? a) (if (gleiche-variable? a v) 1 0))
    ((summe? a) (konstr-summe (ableitung (operand1 a) v) (ableitung (operand2 a) v)))
    ((produkt? a) (konstr-summe (konstr-produkt (operand1 a) (ableitung (operand2 a) v))
                               (konstr-produkt (operand2 a) (ableitung (operand1 a) u))))))
>(ableitung (list '+ 'x 3) 'x))
(list '+ 1 0)
>(ableitung (list '* 'x 'y) 'x) -> (list '* (list '* 'x 0) (list '* 'y 1))≈ 'y
```

Nachteil: Ausdrücke groß und nicht vereinfacht

Vereinfachungsregeln: x*0=0, 0*x=0, 1*x=x, x*1=x

-> Integration in Konstruktoren

```
(define (konstr-summe a1 a2)
  (cond ((and (number? a1 (number? a2)) (+ a1 a2))
         ((number? a1 (if (= a1 0) a2 (list '+ a1 a2)))
         ((number? a2) (= a2 0) a1 (list '+ a1 a2)))
         (else (list '+ a1 a2))))
(define (konstr-produkt a1 a2)
  (cond ((and (number? a1) (number? a2)) (* a1 a2))
         ((number a1) (cond ((= a1 0) 0)
                            ((= a1 1) a2)
                            (else (list '* a1 a2)))
        ((number? a2) (cond ((= a2 0) 0)
                            ((= a2 1) a1)
                            (else (list '* a1 a2))))
  (else (list '* a1 a2))))
```

Num:

```
>(ableitung (list '* 'x 'y) 'x)          -> 'y
```

Implementierungen von Mengen

Mengen: Ansammlung einzelner Objekte (keine Reihenfolge, keine Duplikate)

Datentyp: Operationen: leer, hinzufügen, element?, schnitt, vereinigung

Gesetze: (element? x (hinzufuegen x s) = true
(element? x (vereinigung s1 s2)) = (or (element? x s1) (element? x s2))
(element? x (schnitt s1 s2)) = (and (element? x s1) (element? x s2))
(element? x leer) = false

keine Duplikate: (hinzufuegen x (hinzufuegen x s)) = (hinzufuegen x s)

keine Datenfolge: (hinzufuegen x (hinzufuegen y s)) = (hinzufuegen y (hinzufuegen x s))

16.11.2005

Mengen: als ungeordnete Listen

Idee: Menge ≈ Liste der Elemente ohne Duplikate

```
(define (element? x m)
  (cond ((empty? m) false)
        ((equal? x (first m)) true)
        (else (element? x (rest m)))))
```

equal: Verallgemeinerung von eq? und = auf beliebigen Strukturen

Leere Menge ≈ leere Liste

```
(define leer empty)
```

Hinzufügen: Prüfung ob Element vorhanden

```
(define (hinzufuegen x m)
  (if (element? x m)
      m
      (cons x m)))
```

Schnitt: prüfe für jedes Element einer Menge, ob es in der anderen enthalten ist

```
(define (schnitt m1 m2)
  (cond ((empty? m1) empty)
        ((element? (first m1) m2) (cons (first m1) (schnitt (rest m1) m2)))
        (else (schnitt (rest m1) m2))))
```

Vereinigung: analog

Effizienz:
n: Mächtigkeit der Menge(n)

element?	O(n)	(falls element? nicht vorhanden)
hinzufuegen	O(m)	(wegen element?)
schnitt	$O(n^2)$	(wegen element? -> Rekursion über m1)

=> zu langsam für große Mengen

<u>Verbesserung:</u> Menge als geordnete Listen, d.h. Elemente aufsteigend ordnen
Vor.: es existiert eine Ordnung auf Elementen, hier: Zahlen
hinzufügen: nicht notwendig, am Anfang

```
(define (hinzufuegen x m)
  (cond ((empty? m) (cons x m))
        ((= (first m) x ) m)
        ((< x (first m)) (cons x m))
        (else (cons (first m) (hinzufuegen x (rest m))))))
```

Komplexität in Durchschnitt : $\frac{n}{2}$ Vergleich: Größenordnung: O(n)

element?: analog

Schnitt:
gleichzeitiges Durchlaufen beider Mengen / Listen
Vergleich Anfangselemente: „=“ -> Schnitt, „<“ oder „>“ -> ignoriere kleinstes Element

```
(define (schnitt m1 m2)
  (if (or (empty? m1) (empty? m2))
      leer
      (cond ((= (first m1) (first m2)) (cons (first m1) (schnitt (rest m1) (rest m2)))
            ((< (first m1) (first m2)) (schnitt (rest m1) m2))
            (else (schnitt m1 (rest m2))))))
```

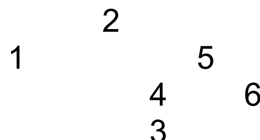
Komplexität:
pro Schritt ein Element herausnehmen => höchstens |m1|+|m2| Schritte
=> O(n)   (statt O(n²)!)

Vereinigung: analog

Mengen als geordnete binäre Bäume

Idee:
linker Teilbaum: enthält nur kleinere Elemente als Wurzel
rechter Teilbaum: enthält nur größere Elemente als Wurzel
Baum für {1,2,3,4,5,6}:

nicht eindeutig:



Vorteil:
(element? x m)

1. x=Wurzel(m): fertig
2. x<Wurzel(m): linker Teilbaum
3. x>Wurzel(m): rechter Teilbaum

=> $\frac{n}{2}$ Elemente

Ein Schritt: $n \to \frac{n}{2}$ => Komplexität: O(log n)

Datenstruktur für Knoten: (eintrag linker-baum rechter-baum), leerer Baum, leere Liste
=> (define-struct baum (eintrag links rechts))
  => Konstruktoren: make-baum, leer
      Selektoren: baum-eintrag, baum-links, baum-rechts
Leere Menge:
(define leer empty)

```
(define (element? x n)
  (cond ((empty? m) false)
        ((= x (baum-eintrag m)) true)
        ((< x (baum-eintrag m)) (element? (x (baum-links m))))
        (else (element? x (baum-rechts m)))))
```

Hinzufügen: Vergleiche x mit Wurzel und füge m linken / rechten Teilbaum ein

```
define (hinzufuegen x m)
  (cond ((empty? m) (make-baum x empty empty))
        ((= x (baumeintag m)) m)
         ((< x (baumeintrag m)) (baumeintrag m) (hinzufuegen x (baum-links m)) (baum-
         rechts m)))
           (else (make-baum (baum-eintrag m) (baum-links n) (hinzufuegen x (baum-rechts
         m))))))
```

Komplexität: analog zu element?: O(log n)

Schnitt / Vereinigung: keine effiziente Möglichkeit, wie bei ungeordneten Listen:
O(n * log n)   n: Durchlauf, log n: element?

Problem:
log n-Laufzeit nur für „ausgewogene“ Bäume
aber: Hinzufügen von 1,2,3,4,5:

```
1
      2
            3
                  4
                        5
```

≈ geordnete Liste

Annahme: zufälliges Hinzufügen

Lösung:
bessere Strukturen, Bäume beim Hinzufügen ausgewogen halten

Anwendungen für Mengen

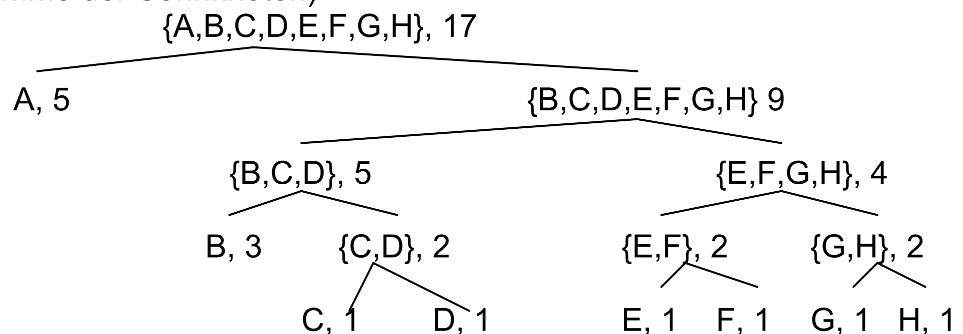
Datenbanken: Elemente: Paare (Schlüssel, Inhalt)
definierte Ordnung: (s1; 1) (s2; 2) <=> s1<s2
=> Mengen der Daten als Bäume
=> schneller Zugriff in logarithmische Zeit

22.11.2005

Beispiel: Huffman-Bäume
Codierung von Daten: 0/1-Folgen
Ascii-Code: ein Zeichen ≙ Folge von 7 Bits (128 verschiedene Zeichen)
=> Nachricht mit 10 Zeichen => Codierung in 70 Bits
Allgemein: Codierung n verschiedener Zeichen => $\log_2 n$ Bits pro Symbol

Nachricht aus 8 verschiedenen Zeichen: A,B,C,D,E,F,G,H 3 Bits ausreichend
z.B. A≙000
B≙001
C≙010
D≙011
E≙100
F≙101
G≙110
H≙111
ABAFBA≙000.001.000.101.001.000 => 18 Bits
„Code fester Länge“
Nachteil: nicht kürzester Code, falls Zeichen unterschiedliche Häufigkeit haben
Idee: häufige Zeichen ≙ kurzer Code =>
seltene Zeichen ≙ langer Code => Code variabler Länge
A≙0
B≙100
C≙1010
D≙1011
E≙1100
F≙1101
G≙1110
H≙1111
=> ABAFBA ≙ 0100011011000 => 13 Bits
Problem: Wo fängt nächstes Zeichen in Bitfolge an?
Lösung: Präfixcode
kein Code (für ein Symbol) ist Präfix (Anfangsstack) eines anderen
Beispiel: s.o.

Idee: Repräsentiere Code als Binärbaum
- Blätter einzelner Zeichen mit Gewichtung (Häufigkeit)
- innerer Knoten: Zeichnungen (Vereinigung aller unterschiedlicher Zeichen) und Gewichtung (Summe der Sohnknoten)



Codierung:

- Code für ein Zeichen:
  Beschriftungen von Wurzel bis zum zugehörigen Blatt

Decodierung:

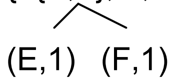
- Folge: Was von Wurzel entspricht der Bitfolge
- Falls Blatt erreicht, Zeichen gefunden

<u>Konstruktion von Huffman-Bäumen</u>

Anfang: nur Blätter mit Häufigkeiten
Schritt: suche Teilbäume mit geringster Gewichtung, vereinige diese zu neuem Baum
Ende: nur noch ein Baum vorhanden.
Beispiel:

{(E,1),(F,1),(G,1),(H,1)} 1. Schritt

{ {E,F}, 2, (G,1), H,1)}

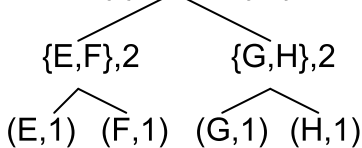


(E,1) (F,1)

{ {E,F}, 2 {G,H}, 2 } 2. Schritt

(E,1) (F,1) (G,1) (H,1)

{ {E,F,G,H},4} 3. Schritt

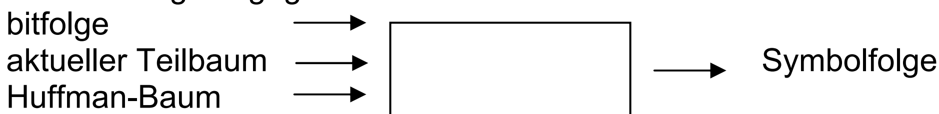


{E,F},2 {G,H},2

(E,1) (F,1) (G,1) (H,1)

Darstellung von Huffman-Bäumen:
Blatt: (define-struct blatt (symbol wichtung))
innere Knoten: (define-struct baum (symbole wichtung links rechts))
symbole: ungeordnete Liste (ok, da alle Symbole verschieden)
wichtung: Knoten oder Blatt

Decodierung mit gegebenem Baum



Implementierung der Baum-Generierung
wichtig: effizientes Auffinden mit geringstem Gewicht
=> Darstellung der Baummenge nach den Gewichten geordnet
initiale Baummenge: wiederholtes Einfügen von Symbolhäufigkeit-Paaren
z.B. (list (make-sh ’A 8) (make-sh ’B 9)…)

2.3 Abstraktionen mit Prozeduren höherer Ordnung

Eine Prozedur höherer Ordnung hat als Argumente oder Ergebnis eine andere Prozedur.

!!! DrScheme: Intermediate Student with lambda !!!

Motivation: häufig Prozeduren mit ähnlichem Schema:
Summe der ganzen Zahlen zwischen a und b

```
(define (ganz-summe a b)
   (if (> a b) (+ a (ganz-summe (+ a 1) b))))
```

Summe aller Quadrate zwischen a und b

```
(define (qu-summe a b)
  (if (> a b) 0 (+ (quadrat a) (qu-summe (+ a 1) b))))
```

Näherungslösung für $\frac{\pi}{8} \frac{1}{1*3}+\frac{1}{5*7}+\frac{1}{9*11}+...$

```
(define (pi-summe a b)
   (if (> a b) 0 (+ (/ 1 (* a (+ a 2))) (pi-summe (+ a 4) b))))
```

gemeinsame Struktur:

```
(define (summe fun naechst a b)
   (if (> a b) 0 (+ (fun a) (summe fun naechst (naechst 1) b))))
```

≈ Mathematik: $\sum_{n=a}^{b} f(b)$ => (define (qu-summe a b) (summe quadrat inc a b))

wobei: (define (inc x) (+ x 1))

23.11.2005

```
(define (summe fun naechst a b)
  (if > a b)
  0
  (+ (fun a (summe fun naechst (naechst a) b))))
```

Beispiel

```
(define (pi-fun x) (/ 1 (* x (+ x 2))))
(define (plus4 x) (+ x 4))
(define (pi-summe a b) (summe pi-fun plus4 a b))
>(* 8 (pi-summe 1 1000))
3,13592
```

Lambda-Abstraktion

obige Definition ist etwas umständlich, da plus4 nur als Parameter notwendig
Besser: anonyme (namenlose) Funktion
statt „plus4“ besser Funktion, die 4 dann addiert
(lambda (x) (+ x 4))

```
(define (pi-summe a b)
  (summe (lambda (x) (/ 1 (* x (+ x 2))))
         (lambda (x) (+ x 4))
         a
         b))
```

Allgemeine Sonderform:

(lambda (x1 … xn) <Rumpf>)
formale Parameter
Wert dieses Ausdrucks Prozedur (ohne Name)

(define (plus4 x) (+ x 4)) |
(define plus 4 (lambda (x) (+ x 4))) | äquivalent
lambda überall anstele von Prozeduramen erlaubt
((lambda (x y) (/ (+ x y) 2)) 2 6) => 4
Mittelwert

lese: „lambda“ als Prozedur
Pragmatik: Falls Prozedur nicht rekursiv und nur an einer Argumentposition benötigt wird, benutze lambda
Geschichte: λ-Kalkül (Church, 1941), exakte Grundlage für Funktionsanwendungen
Grundlange von Lisp

Lokale Definitionen: local

Motivation: lokale Definitionen von global relevanten Unterschieden
(local (<Def1> <Def2>) <Rumpf>)
Definitionen wie im Programm — Ausdruck

Bedeutung:
Führe <Def1>, <Def2> lokal ur Auswertung von <Rumpf> ein.
Beispiel:

```
(local ((define (f x) (+ x 2))
        (define (g y) (* f <) 4)))
  (g 3))
```

Auswertung:

- Strukturierung komplexer Ausdrücke:
  ```
  (define (f x y)
    (local ((define a (+ 1 (x x y))))
            (define b (- 1 y)))
    (+ (* y b) (* a b))))
  ```
  => f(x,y)=y(1-y)+(1+xy)(1-y)
- Einführung lokal relevanter Funktionen
  ```
  (define (pi-summe a b)
    (local ((define (pi-fun x) (/ 1 (* x (+ x 2))))
            (define (plus4 x) (+ x 4)))
    (summe pi-fun plus4 a b)))
  ```

<u>Gültigkeitsbereich, Geltungsbereich</u>
Ausdrucksmenge, in der eine Variablenbindung bekannt ist
Globale Definitionen:

```
(define (f x) <R>)
```

(f x): in Programm gültig
<R>: Geltungsbereich für x

Lokale Definitionen:

```
(define (f x)
  (local ((define (g y) <R>))
    <R'>))
```

(local …): Geltungsbereich von g und x
<R>: Geltungsbereich von y

<u>Lexikalische (statistische) Bindung</u>

- Namen beziehen sich auf textuell nächsten Block, in dem sie gebunden (eingeführt) worden sind
- mögliche gleiche Namen: unterschiedliche Objekte
  ```
  (define (f x)
    (local ((define (g x) x))
      (g x)))
  ```
  unterschiedliche Objekte: lokale Definitionen überschatten globale Definitionen

Beispel:

```
(define (summe fun naechst a b)
  (local ((define (summiere x)
            (if (> x b)
                0
                (+ (fun x) (summiere (naechst x)))))))
```

Pragmatik:
Falls man Hilfsfunktionen benötigt, die global nicht relevant sind, definiere diese mittels „local".

Prozeduren höherer Ordnung: nützlich bei Datenstrukturen: Manipulation
Beispiel: Transformation einer Liste durch Anwendung einer Funktion auf jedes Element

```
>(map-list quadrat (list 1 2 3 4))
(list 1 4 9 16)
```

Implementierung:

```
(define (map-list f l)
  (if (empty? l)
      empty
      (cons (f (first l)) (map-list f (rest l)))))
```

Universelle Berechnungsverfahren

Beispiel: Nullstellenbestimmung durch Intervallhalbierung
gegeben: Funktion f, Punkte a,b mit $f(a)<0<f(b)$
gesucht: Nullstelle von f, d.h. x mit $f(x)=0$
Idee:
Sei x Mittelwert von a und b.

- $f(x)>0$: Suche zwischen a und x
- $f(x)<0$: Suche zwischen x und b

=> Laufzeit wegen Halbierung in jedem Schritt: $O(\log\frac{|b-a|}{T})$ T: Toleranz

```
(define (suche f a b)
  (local ((define x (mittelwert a b)))
    (if (nah-genug? a b)
        x
        (local ((define fx (f x)))
          (cond ((position? fx) (suche f a x))
                ((negativ? fx) (suche f x b))
                (else x))))))

(define (nah-genug? x y) (< (absfx y)) 0.0001))
(define (intervall-halbierung f a b)
  (cond ((and (negative? (f a)) (positive? (f b))) (suche f a b))
        (and (negative? (f b)) (positive? (f a))) (suche f b a))
        (else (error …))))
>(intervall-halbierung sin 2.0 4.0)
2.141113328
>(intervall-halbierung (lambda (x) (- (xx x x x) (* 2 x) 3)) 1 2)
1.8731
```

$f(a) \neq 0 \neq f(b)$

29.11.2005

Prozeduren als Ergebnisse

möglich, falls Ergebnis lambda-Ausdruck oder Funktionsname

Beispiel: Differenzieren von Funktionen: liefert 1. Ableitung (Funktion!) als Ergebnis

$$\frac{d(x^2)}{dx} = 2x$$

Numerische Ableitung: $\frac{df(x)}{dx} = \frac{f(x+dx) - f(x)}{dx}$

Implementierung:

```
(define (ableitung f dx) (lambda (x) (/ (- (f (+ x dx)) (f x)) dx)))

>((ableitung quadrat 0.001) 4)
8.001
```

Beispiel: Komposition von Funktionen $(f \circ g)(x) = f(g(x))$

```
(define (komp f g)
  (lambda (x) (f (g x))))
>((komp quadrat quadrat) 4)
256
>((ableitung (komp quadrat quadrat) 0.001) 3)
108.054
```

Alternative Implementierung von Listen

```
(define (cons x y)
  (local ((define (zuteilen m)
    (cond ((= m 0)
      ((= m 1) y)
      (else (error ’cons „Falscher Wert“)))))
  zuteilen))
(define (first l) (l 0))          (first (cons x y)) = x
(define (rest l) (l 1))           (rest (cons x y)) = y
```

„Wert von cons“ Prozedur, die bei Aufruf mit 0 oder 1 den ersten oder zweiten Parameter zurückliefert

erfüllt Listengesetze

=> Sprache mit Prozeduren höherer Ordnung benötigt keine primitiven Dateistrukturen

=> Daten als Prozeduren wichtiger Programmierstil: Nachrichtenweitergabe, OOP

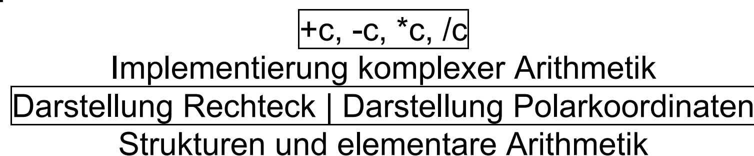
2.4 Daten mit Mehrfachpräsentation

Datenabstraktion: Implementierung mit Abstraktionsbarriere schützen
Vorteil: Implementierung austauschbar

- Prototyp -> effizientere Implementierung (Beispiel Mengen)
- mehrere Implementierungen zur Laufzeit entscheiden, welche Implementierung benutzt wird
  Techniken:
  - geometrische Operatoren
  - „manifeste" Typen: Datenobjekte mit expliziter Typangabe
  - datengesteuerte Programmierung

Beispiel: Komplexe Zahlen
bestehen aus Realteil und Imaginärteil ≈ Punkte in $\mathbb{R}^2$
Rechteckskoordinaten: (x,y) gut für +, -
Polarkoordinaten: (a,φ) gut für *, /

Addition mit Rechteckkoordinaten: $(x_1, y_1) + (x_2 + y_2) = (x_1 + x_2, y_1 + y_2)$
Multiplikation in Polarkoordinaten: $(a_1, \varphi_1) * (a_2, \varphi_2) = (a_1 * a_2, \varphi_1 + \varphi_2)$
Programmstruktur:



Annahme: Gegeben konstr-rechteck, konstr-polar, real-teil. im-teil, abs-wert, winkel

```
(define (+c z1 z2)
  (konstr-rechteck (+ (real-teil z1) (realteil z2)) (+ (im-teil z1) (im-teil z2)))
-c analog
(define (*c z1 z2)
  (konstr-polar (* (abs-wert z1) (abs-wert z2)) (+ (winkel z1) (winkel z2)))
/c analog
```

Zusammenhang beider Darstellungen: $(x, y) \approx (a, w)$:

$$x = a * \cos\varphi$$
$$y = a * \sin\varphi$$
$$a = \sqrt{x^2 + y^2}$$
$$w = \arctan(y, x)$$

Rechteckdarstellung: komplexe Zahl ≈ Struktur mit x und y

```
(define-struct recht (real im))
(define (konstr-rechteck xy ) (make-recht x y))
(define (real-teil-recht z) (recht-real z))
(define (im-teil-recht z) (recht-im ))
(define (abs-wert-recht z) (wurzel (+ (quadrat (recht-real z)) (quadrat (recht-im z))))
(define (winkel-recht z) (atan (recht-im z) (recht-real z)))
(define (konstr-polar-recht a w) (make-recht (* a (cos w)) (* a (sin w))))
```

Polarkoordinatendarstellung: komplexe Zahl ≈ Struktur mit a und φ

```
(define-struct pol (abs phi))
(define (konstr-polar a w) (make-pol a w))
(define (abs-wert-pol z) (pol-abs z))
(define (winkel-pol z) (pol-phi z))
(define (real-teil-pol z) (* (pol-abs z) (cos (pol-phi z))))
(define (im-teil-pol z) (* (pol-abs z) (sin (pol-phi z))))
```

Wunsch: Implementierung von +c, -c, *c, /c unabhängig von Darstellung

Manifeste Typen: Daten enthalten Informationen über Typ der Repräsentation
=> Selektoren wie real-teil implementieren durch real-teil-recht, real-teil-pol
Typ erkennbar durch Strukturprätikate: recht?, pol?
=> Konstruktion in beiden Darstellungen

```
(define (konstr-rechteck x y) (make-recht x y))
(define (konstr-polar a w) (make-polar x y))
```

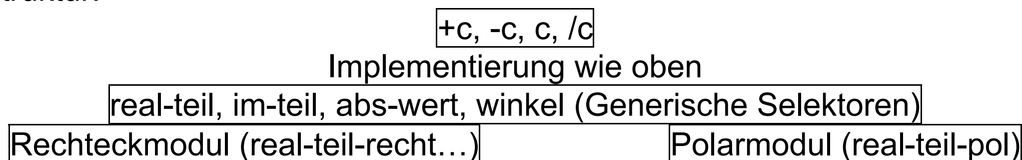
notwendig: generische Selektoren für komplexe Zahlen in beliebiger Darstellung

```
(define (real-teil z)
  (cond ((recht? z) (real-teil-recht z))
           (pol? z) (real-teil-pol z)))
```

=> Fallunterscheidung für alle Typen

analog: im-teil, abs-wert, winkel

Gesamtstruktur:



30.11.2005

Generische Operatoren: arbeiten mit mehreren Datenpräsentationen

Bisherige Implementierung: gen. Operatoren: Fallunterscheidung über alle Repräsentationen
Nachteil: gen. Operatoren müssen alle Repräsentationen „hemmen“
=> neue Darstellung einführen => Code aller gen. Operatoren ändern
Lösung: Datengesteuerte Programmierung: „Daten entscheiden über ihre Bearbeitung“
Beispiel: komplexe Zahl in Rechteckdarstellung liefert Realteil durch Prozedur real-teil-recht
notwendige Bedingung: Zuordnung (generischer Operator, Typ / Darstellung der Daten) -> „echte Prozeduren“
z.B. als Tabelle:

Typ	Recht	Pol
real-teil	real-teil-recht	real-teil-pol
im-teil	im-teil-recht	im-teil-pol
abs-wert	…	…
winkel	…	…

notwendig: Typinformation von Daten als Symbol / Konstante für Tabellenindex

```
(define (typ z)
  (cond ((recht? z) ’recht)
        ((pol? z) ’pol)
        (else (error ’typ „Unbekannter Typ“))))
>(typ (konstr-polar 2 0.5))
’pol
```

Annahme: Gegebene Operationen zu

- Aufbau der Tabelle: (put <typ> <op> <eintrag>)
- Durchsuchen der Tabelle (get <typ> <op>)
  Ergebnis: Eintrag oder empty, falls kein Eintrag existiert

  <typ>: Typsymbol
  <op>: Symbol des gen. Operators
  <eintrag>: Bearbeitungsprozedur

Implementierung (Kapitel 3)

Aufbau konkrete Tabelle

```
(put ’recht ’real-teil real-teil-recht)
(put ’pol ’real-teil real-teil-pol)
```

1. Einträge sind Prozeduren
2. Hinzufügen einer neuen Darstellung: Modul implementieren und put-Operation

Ausführen einer gen. Operation für ein Objekt

```
(define (op-ausfuehren op obj)
  (local ((define proz (get (typ obj) op)))
    (if (not (empty? proz))
      (proz obj)
      (error ’op-ausfuehren „Operator für den Typ undefiniert“))))
(define (real-teil z) (ob-ausfuehren ’real-teil z))
```

Allgemein: Typprüfung + Aufruf einer passenden Prozedur – „dispatching on type“

Nachrichtenweitergabe: Intelligente Datenobjekte“
Objekte enthalten Wissen zur Bearbeitung
Datenobjekte enthält Zuordnung gen. Operatoren -> Implementierung (x Tabellenspalte)

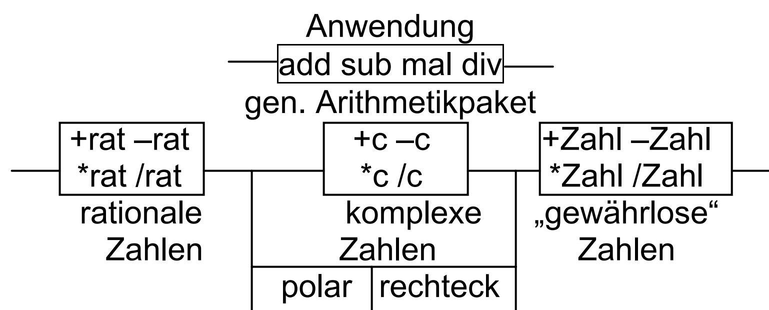
Beispiel: Rechteckobjekte
```
(define (konstr-rechteck x y)
  (local ((define (zuteilen n)
        (cond ((eq? n ’real-teil) x)
              ((eq? n ’im-teil) y)
              ((eq? n ’abs-wert) (wurzel (+ (quadrat x) (quadrat y))))
              ((eq? n ’winkel) (atan y x))
              (else (error ’rechteck „Unbekannte Operation“)))))
  zuteilen)     ; Analogie zur alternativen Listenimplementierung

(define (op-ausfuehren op obj) (obj op)))
```

Programmiertechnik: Nachrichtenweitergabe: message passing
+ Zustände (Kapitel 3) ≈ objektorientierte Programmierung

2.5 Systeme mit generischen Operatoren

Beispiel: Generisches Arithmetikmodul
Operatoren für Zahlen aller Art: reelle Zahlen, rationale Zahlen, komplexe Zahlen

Struktur:



Vorgehen analog zu bisher, mit Typinformation an den Daten (’rat, ’komplex, ’zahl)
z.B. Modul für gewöhnliche Zahlen
```
(define (+zahl z1 z2) (konstr-zahl (+ (zwert z1) (zwert z2))))
```

Darstellung dieser Zahlen mittels Strukturdefinition:
```
(define-struct zahl (wert))
(define (konstr-zahl n) (make-zahl n))
(define (zwert z) (zahl-wert z))
```

Nun: Verbindung gen. Operatoren (add, sub, ..) mit Zahl-Implementerung
Hinweis: Tabellentechnik: Erweiterung typ-Prozedur: (cond … ((zahl? z) zahl)

```
(put ’zahl ’add +zahl)
(put ’zahl ’sub –zahl)
…
(define (add x y) (op-ausfuehren-2 ’add x y))
(define (sub x y) (op-ausfuehren-2 ’sub x y))
…
```

op-ausfuehren-2: wie op-ausfuehren, aber mit 2 Argumenten

```
(define (op-ausfuehren-2 op a1 a2)
  (if (eq? (typ a1) (typ a2))
     (local ((define proz (get (typ a1) op)))
        (if (not (empty? proz))
           (proz a1 a2)
           (error … „Operanden für den Typ undefiniert“)))
     (error … „Operanden haben ungleichen Typ“)))
```

kompleze Zahlen hinzufügen:
analog zu gewöhnlichen Zahlen:

```
(define-struct komplex (wert))
(define (konstr-komplex z) (make-komplex z))
(define (kwert z)) (komplex-wert z))
(define (+komplex z1 z2) (konstr-komplex (+c (kwert z1) (kwert z2))))
…
```

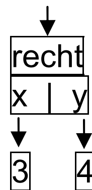
Zuordnung zu generischen Operatoren:
Erweitere typ-Prozedur: (cond… ((komplex? z) ’komplex)

```
(put ’komplex ’add +komplex)
…
```

6.12.2005

Interessant: <u>2-stufiges Typsystem</u>
Konstruktion: komplexe Zahl in Rechteckkoordinaten
(konstr-komplex (konstr-rechteck 3))

-> Intern: komplex-wert <- für generisches Arithmetikmodul



<- für Komplexmodul mit Mehrfachdarstellung

Anwendung
↓ „Abschriften der Strukturen"
Implementierung

Anwendung
↓ Zufügen der Strukturen
Implementierung

Operanden unterschiedlicher Typen

Beispiel: Addition 3 und 2+5i -> Error!
Mögliche Lösung: Operatoren für gemischte Operanden,
z.B. +zahl-komplex, +rational-komplex
und 3-dimensionale Tabelle: Typ x Typ x Opname -> Prozedur
Nachteil: Aufwand n verschiedener Typen

Allgemein: Operator mit n Argumenten $\Rightarrow n^m$ Versionen
Besser: Typanpassung: Versuche Typ eines Operanden in den anderen zu überführen

Beispiel: Zahl -> komplexe Zahl
(define (zahl->komplex x) (konstr-komplex (konstr-rechteck (zweit x) 0)))
Datengesteuerte Programmierung: alle möglichen Typanpassungen in spezielle Tabelle
z.B. (put-typanpassung 'zahl 'komplex zahl->komplex)
Quelltyp Zahltyp Anpassungsprozedur

nun: op-ausfuehren-2 ändern:

```
(define (op-ausfuehren-2 op a1 a2)
  (local ((define t1 (typ a1)) (define t2 (typ a2)))
    (if (eq? t1 t2)
       (local ((define proz (get t1 op)))
         (if (not (empty? proz)) (proz a1 a2) (error … „Operator unbekannt")))
       (local ((define t1->t2 (get-typanpassung t1 t2))
               (define t2->t1 (get-typanpassung t2 t1)))
       (cond ((not (empty? t1->t2)) (op-ausfuehren-2 (t1->t2 a1) a2))
             ((not (empty? t2->t1)) (op-ausfuehren-2 a1 (t2->t1 a2)))
             (else (error 'op-ausfuehren-2 „Operanden haben unanpassbaren Typ")))))))
```

Vorteil: n Typen => n Versionen jedes generischen Operators (statt n²)
Nachteil: n² (maximal) Anpassungsprozeduren
vorstellbar: Objekte mit Typen t1 und t2, aber weder t1->t2 noch t2->t1 möglich
evtl. gemeinsamer „Obertyp" t3 mit t1->t3 mit t2->t3

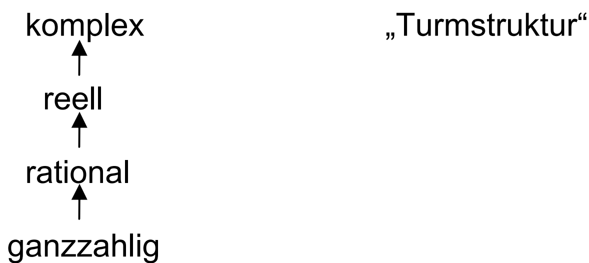
Betrachte Typhierarchien
t1 Untertyp von t2

- „Jedes t1-Objekt ist auch t2-Objekt"
- „Jeder auf t2-Objekte anwendbare Operator ist auch auf t1-Objekte anwendbar"
- „t1-Konzept ist Spezialfall von t2-Konzept"

Notation: t2

t1
auch: t2 Obertyp von t1

Typhierarchie für Zahlen:


komplex „Turmstruktur"
reell
rational
ganzzahlig

wichtig: Untertyp immer in Obertyp anpassbar

Typanpassung nur noch: t-> „nächsthöherer Typ von t" (hier nur noch 3 Prozeduren)
Umwandlung: ganzzahlig -> komplex: schrittweise ganz-> rational -> reell -> komplex

Änderungen in op-ausfuehren-2 bei verschienenen Typen:
erhöhe schrittweise niedrigeren Typ, bis beide gleich sind (-> Übung)

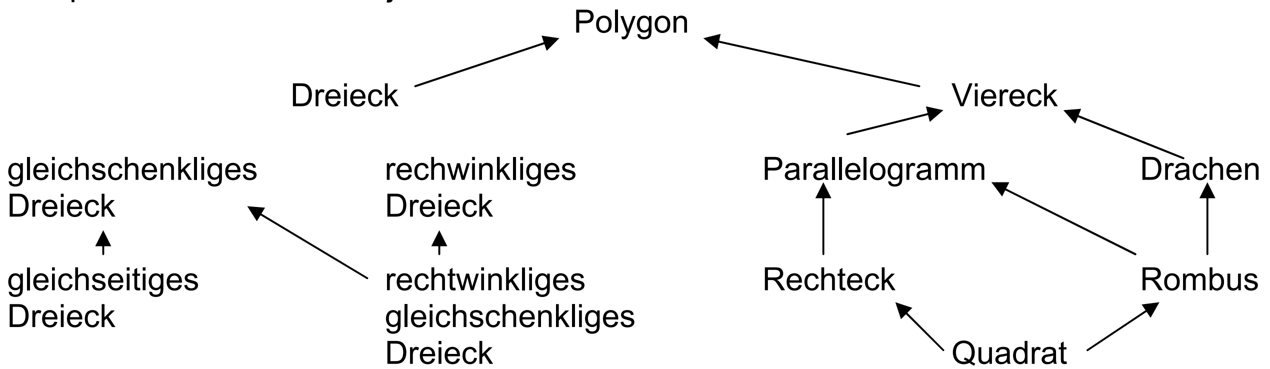
weiterer Vorteil: Vererbung von Operationen
Falls Operation auf Typ undefiniert: statt „error…": erhöhe Typ der Argumente und versuche erneut, Operation anzuwenden
z.B. Wurzel auf ganze oder rationale Zahlen

weiter: Erniedrigen von Typen

- Umwandlung von Objekten in niedrigerem Typ (falls möglich)
- z.B. Addition von 2-3i und 4-3i -> ganze Zahl 6 (statt 6+0i)
- notwendig: Kriterium: ist Objekt von niedrirerem Typ? -> Übung

Komplexere Typhierarchien
Turmstruktur ideale Hierarchie (einfach), allerdings i.A. komplexer

Beispiel: Geometrische Objekte:



Probleme:
„Mehrfachvererbung“: kein eindeutiger Typ für Erhöhung
Mehrere Untertypen: keine eindeutige Erniedrigung

Typanpassung: erhöhe Operanden auf „kleinsten gemeinsamen Obertyp“

Beispiel: gleichseitiges Dreieck + Quadrat -> Polygon
notwendig: Suche in Typgraph (effiziente Implementierung: objektorientierte Sprachen)

2.6 Exkurs: Beweistechniken für Programme mit Datenstrukturen

Montivation: ADT: trenne Schnittstelle von Implementierung
Bedeutung ADT: Festlegung durch Gleichungen => Implementierung korrekt: erfüllt Gleichungen
Gleichungen enthalten Variablen: Nachweis der Gleichungen für beliebig viele Terme

Beobachtung: Terme sind regelmäßig aufgebaut (über Konstruktoren)

analog zu Zahlen
natürliche Zahlen:

- 0 ist natürliche Zahl
- n natürliche Zahl => n+1ist natürliche Zahl

Liste:

- empty ist Liste
- l Liste => (cons e l) Liste

Beweisprinzip über natürliche Zahlen: vollständige Induktion
P: Prädikat (Aussage) über natürliche Zahlen
falls:

- P(0) wahr
- P(n) wahr impliziert P(n+1) wahr

denn ist P(n) wahr für alle $n \in \mathbb{N}$

Vollständige Induktion: (P(0) und (P(n)=>P(n+1)) => P(n) $\forall n \in \mathbb{N}$

Anders für Datenstrukturen: Strukturelle Induktion
Falls für jedes n-stellige Funktionssymbol $f \mid n \in \Sigma$ gilt
Falls P(t_1),…,P(t_n) wahr, dann ist auch P(f($t_1,\ldots,t_n$)) wahr (*)
dann gilt für alle Terme $t \in T_\Sigma(\varnothing)$ P(t)
(*) ist Induktionsbasis n=0 und Induktionsschritt für n>0
Falls Σ Konstruktoren einer Datenstruktur enthält: Methode, um Aussagen für alle konstruierten Terme zeigen zu können

Beispiel: Längenberechnung von Listen ist korrekt
```
(define (length l)
  (if (empty? l) 0 (+1 (length (rest l)))))
```

Behauptung:
Falls l Liste mit n Elementen ($n \geq 0$), dann gilt (length l) =>*n
=> P(l)

Konstruktoren für l: empty/0, cons/2

1. Fall: l=empty (d.h. n=0): (length empty) => (if (empty? empty) 0 (+ 1 …))
   => (if true 0 …)
   => 0
2. Fall: l=(cons e l'), l' Liste mit n Elementen ($n \geq 0$), d.h. l Liste mit n+1 Elementen
   zu zeigen: falls (length l')=>*n, dann gilt (length l)=>*n+1
   P(l') = P(l)
   (length l)=(length (cons e l'))
   => (if (empty? (cons e l')) 0 (+ 1 (length (rest (cons e l')))))
   => (if false 0 (+ 1 (length (rest (cons e l')))))
   => (+ 1 (length (rest (cons e l'))))
   => (+ 1 (length l')) =>' (+ 1 n) = n+1

Beispiel: Korrektheit von append
```
(define (append l1 l2)
  (if (empty? l1) l2 (cons (first l1) (append (rest l1) l2)))))
```
Behauptung: falls l1=(list $x_1 \ldots x_n, y_1 \ldots y_m$), dann gilt
(append l1 l2)=>* (list $x_1 \ldots x_n, y_1 \ldots y_m$) ($n, m \geq 0$)
=> P(l1) (Induktion über l1)

1. Fall: l1=empty (d.h. n=0)
   (append l1 l2)=(append empty l2) => (if (empty? empty) l2 (…))
   => => l2 = (list $y_1 \ldots y_m$)
2. Fall: l1= (cons $x_1 \ldots x_s$), x_s =(list $x_2 \ldots x_n$)
   Annahme: (append x2 l2) =>* (list $x_2 \ldots x_n, y_1 \ldots y_m$)
   (append l1 l2)=(append (cons $x_1 \ldots x_s$)) l2 (cons …))
   => => (first (cons $x_1 \ldots x_s$)) (append (rest (cons $x_1 \ldots x_s$)) l2))
   =>* (cons x1 (append xs l2))
   =>* (list $x_1 \ldots x_n, y_1 \ldots y_m$)

=> Aussage für alle Listen l1 und l2

3. Modularität, Objekte, Zustände

SW-Erstellung:
Orientierung an Struktur des Anwendungsproblems
+ „geeignete" Organisationsstruktur (Module, Abstraktion)

bisher: math. Berechnungen: Prozeduren
datenintensive Anwendungen, hierarchische Datenstruktur + Datenabstraktion
häufig: reale Welt: komplexe Objekte mit „Geschichte" (Bankkonto, Flugbuchungen, …)

3.1 Zuweisungen und lokale Zustände

Konto: (abheben 100) -> je nach der Vorgeschichte!

Abstraktion der Geschichte auf einen veränderbaren Zustand: Kontostand

Beispiel: Abheben
Falls möglich, Rückgabe: aktueller Kontostand, sonst entsprechende Nachricht

```
>(abheben 80)
80
>(abheben 50)
30
>(abheben 50)
Deckung nicht ausreichend!
```

bisher:
Prozedur = Funktion mit eindeutiger Antwort
Variable = eindeutiger Wert

neu:
Änderung von Variablenarten: Sonderform
(set! <name> <neuer-wert>)
<name>: definiertes Symbol (mittels define)
<neuer-wert>: Ausdruck

DrScheme: Advanced Student!!!

Effekt:

1. werte <neuer-wert> aus
2. verändere Definition von <name> zu berechnen: Wert (d.h. neues define)
3. Ergebnis: undefiniert

```
(define kontostand 160)

(define (abheben betrag)
  (if (>= kontostand betrag)
    (begin (set' kontostand (- kontostand betrag))
      kontostand)
    „Deckung nicht ausreichend"))
```

Sonderform: (begin $<a_1><a_2>\ldots<a_n>$)

1. werte nacheinander Ausdrücke $<a_1><a_2>\ldots<a_n>$ aus (bilden: Auswertung von Argumenten beliebig)
2. Ergebnis der Sonderform: Wert von $<a_n>$

Nachteil: herleitend überall bekommt und veränderbar
Lösung: definiere kontostand lokal in abheben

```
(define neu-abheben
  (local ((define kontostand 160))
    (lambda (betrag) (if …wie oben…))))
```

- Kontostand lokal
- gleiches Verhalten wie abheben

Nachteile:
- fixer Anfangskontostand
- ein festes Konto

=> Abhebungsprozessoren für beliebige Konten und Beträge

```
(define (konstr-abheben wert)
  (local ((define kontostand wert))
    (lambda (betrag)                    ; Ergebnis: Prozedur zum Abheben
      (if … wie oben…))))
```

zwei Konten:

```
(define k1 (konstr-abheben 100))
(define k2 (konstr-abheben 100))
```

```
>(k1 50)
50
>(k2 70)
30
>(k2 40)
Deckung nicht ausreichend
>(k1 40)
10
```

(set! variable wert) (begin $a_1, ..., a_n$)
Lokale Zustände müssen modelliert werden, das Prinzip dazu ist das Prinzip der Nachrichtenweitergabe (=> OOP)

Wir kommen wieder auf unser Bankkonto als Beispiel zurück, diesmal allerdings mit der Möglichkeit eine Einzahlung zu tätigen (neben dem Abheben).
Zunächst brauchen wir einen Konstruktor, der ein neues Konto erzeugt:

```
(define (konstr-konto wert) ;Erzeuge neues Konto
  (local ((define kontostand wert)
          (define (abheben betrag)
            (if (>= kontostand betrag)
                (begin
                  (set! kontostand (-kontostand betrag))
                  kontostand)
                (error ’Deckung: "Deckung nicht ausreichend"’))
          (define (einzahlen betrag)
                (begin
                  (set! kontostand (+ kontostand betrag))
                  kontostand))
          (define (zuteilen m)
                (cond ((eq? m ’abheben) abheben)
                      ((eq? m ’einzahlen) einzahlen)
                      (else (error ’konto "Unbekannte Forderung"))))))
  zuteilen))
```

```
> (define kto (konstr-konto))
> ((kto ’abheben) 120)
Deckung nicht ausreichend

> ((kto ’einzhalen) 50)
150

> ((kto ’abheben) 120)
30
```

Nun wollen wir das zuteilen für den Anwender verbergen:

```
(define (abheben k b) ((k ’abheben) b))

(define (einzahlen k b) ((k ’einzahlen) b))
```

Graphische Darstellung der Struktur von Objekten mit Nachrichtenweitergabe:
Angabe von Klassendiagrammen: (-> OOP)

1. Name
2. lokale Werte
3. lokale Prozeduren

1. Konto
2. Kontostand
3. (abheben betrag)
   (einzahlen betrag)

Probleme der Zuweisung

Das Substitutionsmodell ist nun nicht mehr anwendbar!

(define kontostand 130)
-> Namen "kontostand" durch 130 ersetzen

(+ e kontostand) (+ 20 130)
-> haben die gleichen Werte, falls der Wert von e 20 ist

(+ (abheben 110) kontostand)
-> (+2020) bei "links-rechts-Anwendung"
-> (+20130) bei "rechts-links-Anwendung"

Somit sind bei set! die Variablen keine Namen für einfache Werte, sondern es sind Namen für Orte, wo Werte gespeichert werden.
Weiterhin spielt jetzt die Auswertungsreihenfolge eine Rolle, die Reihenfolge der Auswertung legt man mit (begin…) fest.

Es ist nun unzulässig Ausdrücke durch Werte in einem Programm zu ersetzen:

```
(local ((define x (abheben 100))
  (+ x 20))
```

Den Ausdruck (+ x 20) haben wir früher durch den Ausdruck bzw. Wert 40 ersetzt.
Dieses ist aber unzulässig(!), da man nicht zu 100% sagen kann, dass die Rechnung bei jedem Aufrufen den Wert 40 ergibt.

Referentielle Transparenz

Bei der referentiellen Transparenz handelt es sich um eine Eigenschaft von Programmiersprachen.
"Gleiches durch Gleiches" ist zulässig, ohne dass die Werte verändert werden (das Substitutionsmodell ist somit anwendbar).
Die referentielle Transparenz wird durch (set!) "zerstört".
Ohne die ref. Transparenz ist "Gleichheit" schwieriger.
Bei (set!) gibt es zwei Gleichheiten

- Wertgleichheit
- Objektgleichheit, Objektidentität

(define peter-kto (konstr-konto 100))
(define paul-kto (konstr-konto 100))
=> es handelt sich hier um verschiedene Konten

(define peter-kto (konstr-konto 100))
(define paul-kto peter-kto)
=> es handelt sich hier um ein Konto mit zwei Namen

Vorzüge der Zuweisung

Die Modularität wird durch Objekte mit lokalem Zustand gewährleistet.

Beispiel: Zufallsgenerator
Der Zufallsgenerator soll eine Reihe von zufälligen (statisisch gleich verteilten) Werten ausgeben.
Annahme: (zufall-aktuell x) erzeugt ein zufälliges nächstes Element
$x_1 = x_1 x_2 = (zufall - aktuell\ \ x_1)$
$x_3 = (zufall - aktuell\ \ x_2)$
usw.

=> $x_1, x_2, x_3, ..., x_n$ ist eine zufällige Folge.

```
(define zufall
  (local ((define x zufall-init))
         (lambda ()
         (begin (set! x (zufall-aktuell x))
                x))))
```

Der Vorteil ist, dass man (zufall) von allen Programmteilen unabhängig aufrufen kann.

Anwendung: Monte-Carlo-Simulation
- große Menge von Experimenten
- zufällige Auswahl der Stichproben
- Tabellierung der Ergebnisse => Die Wahrscheinlichkeiten erlauben Schlussfolgerungen

Hier:
Die Wahrscheinlichkeit, dass zwei zufällige Zahlen keinen gemeinsamen Teiler >1 haben, ist $\frac{6}{\pi^2}$. (Césaro-Test)

Die Nährung von π durch die Monte-Carlo-Simulation lautet $p = \frac{6}{\pi^2} \Rightarrow \pi = \sqrt{\frac{6}{p}}$.

```
(define (schaetzwert-pi versuche)
        (sqrt (/ 6 (monte-carlo versuche cesaro-test))))

(define (cesaro-test)
        (= (ggt (zufall) (zufall)) 1))

(define (monte-carlo veruche experiment)
        (local ((define (iter versuche-uebirg versuche-erfolgreich)
                        (cond ((= versuche-uebrig 0)
                                (/ versuche-erfolgreich versuche))
                               ((experiment) (iter (- versuche-uebrig 1)
                                                   (+ versuche-erfolgreich 1)))
                        (else (iter (- versuche-uebrig 1)
                                    versuche-erfolgreich)))))
        (iter versuche 0)))
```

Ohne (zufall) aber mit (zufall-aktuell):

```
(define (schaetzwert-pi versuche)
        (dqrt (/ 6 (zufalls-ggt-test versuche zufall-init))))

(define (zufalls-ggt-test verushce anfangs-c)
        (local
              ((define (iter versuche-uebrig versuche-erflogreich x)
                        (local
                               ((define x1 (zufall-aktuell x))
                                (define x2 (zufall-aktuell x1))))
                        (cond ((= versuche-uebrig 0)
                                (/ versuche-erfolgreich versuche))
                               ((= (ggt 1 x2) 1)
                                (iter (- versuche-uebrig 1)
                                      (+ versuche-erfolgreich 1)
                                      x2))
                                (else (iter (- versuche-uebrig 1)
                                            versuche-erfolgreich
                                            x2)))))
        (iter versuche 0 anfangs-x)))
```

Hierbei handelt es sich um kein allgemeines (monte-carlo).
Der Zustand des Zufallsgenerators muss weitergereicht werden, das bedeutet, dass es zu einer Verquickung vom mitzählen der Ergebnisse und Zufallszahlerzeugung kommt.
=> Zustandsänderung kann Modularität erhöhen.

14.12.2005

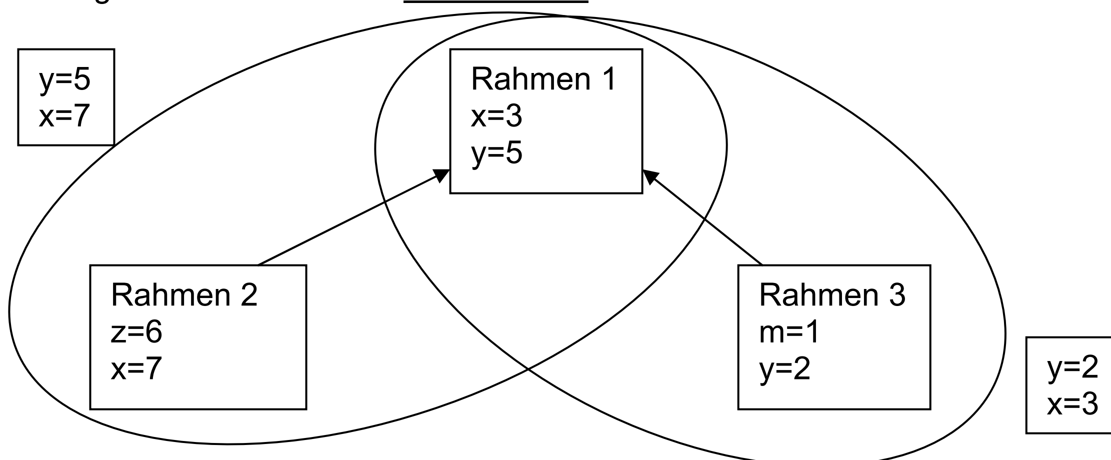
3.2 Das Umgebungsmodell

Substitutionsmodell: Variable ≈ Name für Wert
Jetzt: Variable ≈ Name für Ort, wo Werte gespeichert werden
Struktur notwendig, wegen lokaler Definitionen

Umgebung: Folge von (Bindungs)Rahmen (frames)
Rahmen: Tabelle von Bindungen + Verweis auf zugehörige Umgebung (außer globaler Rahmen)
Bindungen: Zuordnung: Variablen-> Wert

Wert einer Variablen bezüglich einer Umgebung
Wert derjenigen Bindung der Variablen, die sich im ersten Rahmen befindet, der eine Bindung für die Variable enthält.

keine Bindung => Variable ungebunden (≈Fehler)
Bindungen können einander überschatten



Umgebung bestimmt Kontext einer Auswertung
-> Zuordnung von Variablen -> Werten

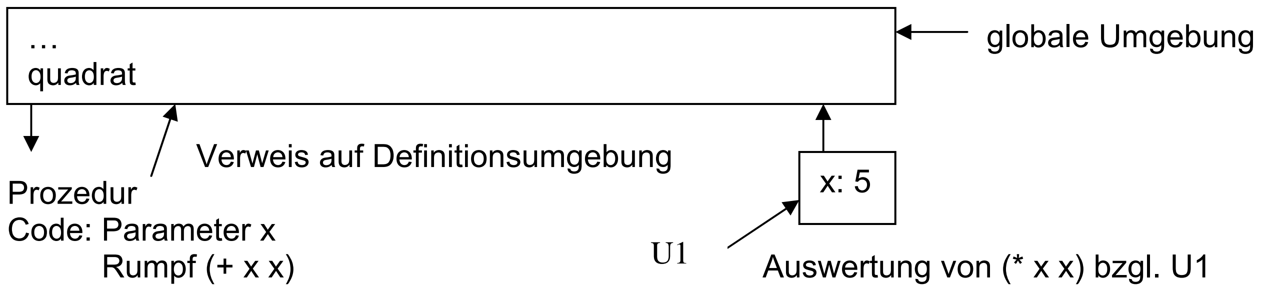
Auswertungsregeln Wert einer Kombination bezüglich einer Umgebung
1. Werte Teilausdrücke bezüglich der Umgebung aus (beliebige Reihenfolge!)
2. Werte Wert des Operatoraustauschs (Prozedur) auf Werte des Operandenausdrücke an

Schritt 2: anders als in Substitutionsmodellen

Prozedur ≈ Code (Parameter + Rumpf) + Zeiger auf (Definitions)Umgebung

Beispiel:
(define (quadrat x) (* x x))
äquivalent zu:
(define quadrat (lambda (x) (* x x)))

Auswertung (quadrat 5) bezüglich globaler Umgebung:



Anwenden einer Prozedur auf Argumente

1. Konstruiere neuen Rahmen mit zugehöriger Umgebung = Definitionsumgebung der Prozedur
2. Trage dort Bindungen Parameter -> aktuelle Werte ein
3. Werte Prozedurrumpf bezüglich der neuen Umgebung aus

Auswertung eines lambda-Ausdrucks bezüglich einer Umgebung

Erzeuge Prozedurobjekt:
- Parameter + Rumpf an lambda-Ausdruck
- Verweis auf aktuelle Umgebung

Auswertung define bezüglich einer Umgebung

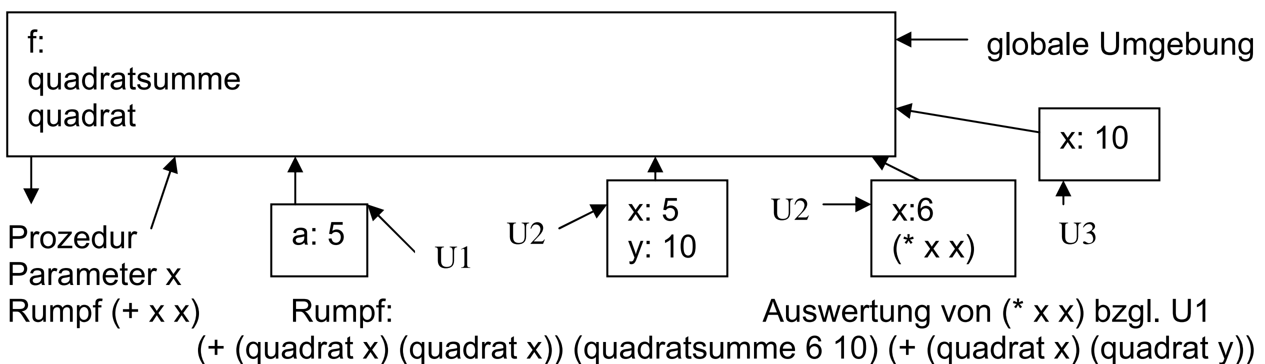
Werte define-Ausdruck aus und erzeuge neue Bindung Variable -> Wert in aktueller Umgebung

Auswertung (set! x a) bezüglich einer Umgebung

1. Suche ersten Rahmen in aktueller Umgebung mit Bindung für x (nicht vorhanden => Fehler)
2. Ändere Wert für x zu Wert von a

Beispiel: Auswertung einfacher Prozeduren
(define (quadrat x) (* x x))
(define (quadratsumme x y) (+ (quadrat x) (quadrat y)))
(define (f a) (quadratsumme (+ a 1) (* a 2)))

Auswertung der define: Erzeugung neuer Prozedurobjekte in globaler Umgebung



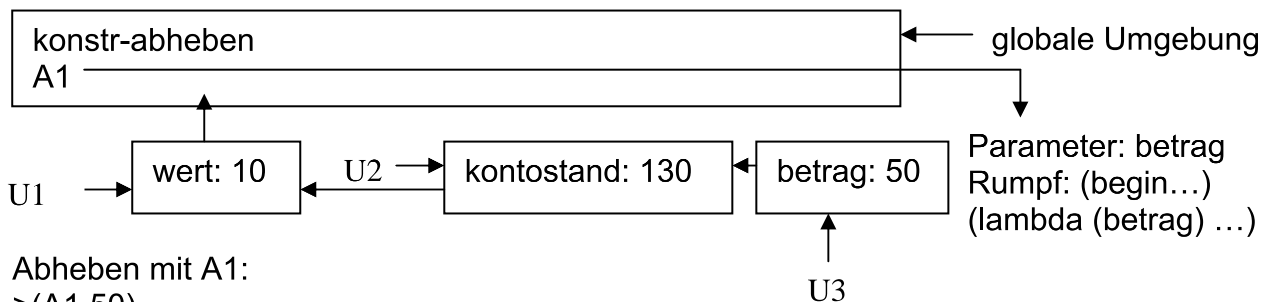
Lokale Zustände und Prozeduren

Beispiel: „Abheben-Prozedur“ (vereinfacht)

```
(define (konstr-abheben wert)
  (local ((define kontostand wert))
    (lambda (betrag)
      (begin (set! kontostand (- kontostand wert)
        (kontostand)))
```

Definition => Eintrag in globale Umgebung

(define A1 (konstr-abheben 130))



Abheben mit A1:
>(A1 50)
-> 1. Neue Umgebung U3 für Ausdruck
   2. Rumpf in U3

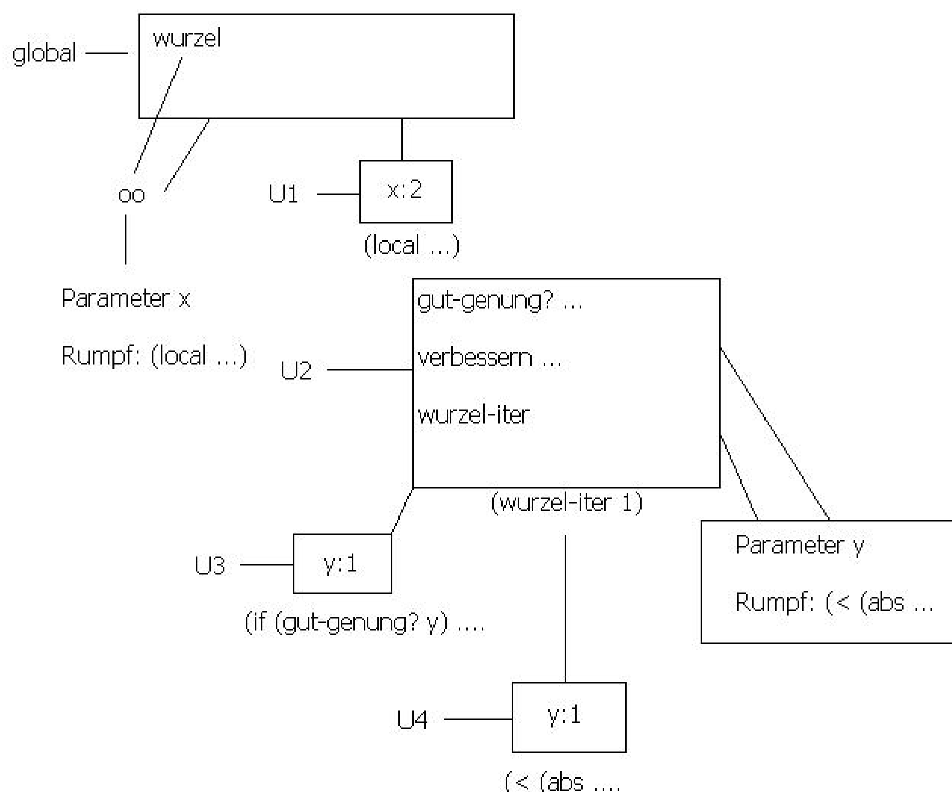
Auswertung von (local (<Definitionen> <Ausdruck>) in einer Umgebung
1. Erzeuge neue Umgebung mit neuen Rahmen, zugehöriger Rahmen: aktuelle Umgebung
2. Trage <Definitionen> in neuen Rahmen ein
3. Werte <Ausdruck> in neuer Umgebung aus

=> Lokaler Zustand von A1 in U2 gespeichert
=> Erzeugung eines neuen Kontos: neue Umgebung für Kontostand anlegen
>(define A2 (konstr-abheben 100))

Die Umgebung ist eine Folge von Rahmen. Die Rahmen werden mit dem Aufruf einer Prozedur + local eingeführt und bei einem Rahmen handelt es sich um eine Folge von Bindungen, wobei es sich bei einer Bindung um eine Zuweisung eines Wertes in eine Variable handelt.

Lokale Definition:
Ein Beispiel zur Berechnung der Wurzel mit Hilfe von lokalen Definitionen:

```
(define (wurzel x)
  (local
    ((define (gut-genung? y)
              (< (abs (- (quadrat y) x)) 0.001))
     (define (verbessern y)
             (mittelwert y (/ x y)))
     (define (wurzel-iter y)
  (if (gut-genung? y)
      y
      (wurzel-iter (verbessern y)))))
  (wurzel-iter 1)))
```



Umgebung bei der Ausführung von (wurzel 2)

Das Umgebungsmodell erklärt den Effekt von lokalen Definitionen:
- lokale Namen in globaler Umgebung unsichtbar
- innerhalb lokaler Prozeduren: Parameter der umgebenden Prozedur sichtbar (z. B. ”x“)
- keine Namenskonflikte bei den Parameternamen auf gleicher Ebene (z. B. ”y“)

3.3 Veränderbare Datenstrukturen

Es ist sinnvoll (z. B. put bei Tabellen) Teile von Datenstrukturen zu verändern.

veränderbare Datenobjekte (mutable data objects): Datenobjekt, für die Mutatoren definiert sind
Mutator: (”Veränderer“) für Struktur: Prozedur, die Komponenten abändert

Listenmutationen

```
(set-first! l e)
(det-rest! l r)
```

```
> (define l (list 1 2))
> (set-first! l 3)
> l
(list 3 2)
> (set-rest! l (list 4 5))
> l
(list 3 4 5)
```

Mutator: (append! l1 l2) Verbinde Liste l1 (nichtleer!) mit l2, d.h. wie append, aber l1 abändern.

```
(define (letztes-cons l)
  (if (empty? (rest l))
     l
     ((letztes-cons (rest l)))))

(define (append! l1 l2)
  (begin
    (set-rest! (letztes-cons l1) l2)
    l1))
```

```
> (define l1 (list 1 2))
> (define l2 (list 3 4))
> (append! l1 l2)
(list 1 2 3 4)
> l1
(list 1 2 3 4)
```

Bei dem append gibt es allerdings ein Problem, so dass man append nur eingeschränkt nutzen kann:

```
> (append! l1 l2)
(list 3 4 3 4 3 4 3 4 3 4 3 4 3 4 3 4 3 4 3 4 . . . . . . . . .)
```

Mutatoren für beliebige Strukturen (DrScheme)

```
(define-struct s (k1 ... kn))
```

Eingeführte Operatoren:

1. (make-s x1 ... xn)
2. (s-ki x)
3. (s? x)
4. (set-s-ki! x xi)

Die Punkte 1 bis 3 sind schon bekannt. Der Punkt ist neu. Dabei handelt es sich um eine Mutator.

```
(define-struct rat (zaheler nenner))
> (define eindrittel (make-rat 1 3))
> (set-rat-zaehler! eindrittel 2)
> eindrittel
(make-rat 2 3)
```

Structure Sharing und Identität

```
>(define l (list 1 2))
>(define l1 (cons l l))
```

-> „structure sharing" der Komponenten der Komponenten von l1

```
>(define l2 (cons (list 1 2) (list 1 2)))
```

Atomare Werte (Zahlen) nur einmal repräsentiert (kein Problem, da nicht veränderbar)
Auswertung von cons -> neue cons-Struktur anlegen
l1 und l2 sind „gleiche" Listen: (list (list 1 2) 1 2)

Unterschied bei Mutatoren:

```
(define (set-ff-9! l)
  (begin (set-first! (first l) 9)
         l))
>(set-ff-9' l2)
(list (list 9 2) 1 2)
>(set-ff-9! l1)
(list (list 9 2) 9 2)     geordnet wegen structure sharing
```

Sharing kann mit eq? entdeckt werden:
(eq? x y) wahr genau dann, wenn x und y identische Objekte (Verweise) sind

```
>(eq? (first l1) (rest l1))
true
>(eq? (first l2) (rest l2))
false
```

Mutation = Zuweisung

wegen Sharing-Problematik: Mutation weglassen?
keine Lösung, da Mutation mittels Zuweisung implementierbar

Implementierung analog zu Kapitel 2.2 Nachrichtenweitergabe

```
(define (cons x y)
```

```
  (local ((define f x)
          (define r y)
          (define (set-x! v) (set! f v))
          (define (set-y! v) (set! r v)))
          (define (zuteilen m)
            (cond  ((eq? m ’first) f)
                   ((eq? m ’rest) r)
                   ((eq? m ’set-first!) set-x!)
                   ((eq? m ’set-rest!) set-y!
                   (else error …)))))
  zuteilen))

(define (first l) (l ’first))
(define (rest l) (l ’rest))
(define (set-first! l e) ((l ’set-first!) e))
(define (set-rest! l r) ((l ’set-rest!) r))
```

=> nur mit set! alle Probleme der Mutation präsent

daher: rein funktionale Sprachen haben keine Zuweisung

Mutatoren:
setfirst!
setrest!

Alltemein:
(define-struct s ($k_1..k_n$))
=> set-s-k_i! (i=1,…,n)

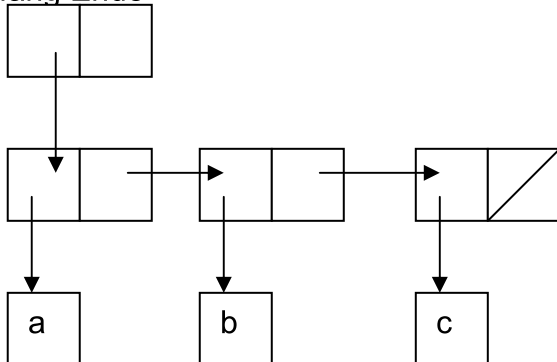
Warteschlange („queue“)
- Hinzufügen am Ende
- Entfernen am Anfange

FIFO (First In – First Out) (Gegensatz zu LIFO – Last In – Last Out) – Stack, Keller
Warteschlangeninhalt
<>
Hinzufügen a: <a>
Hinzufügen b: <a b>
Entfernen: <b>
Hinzufügen c: <b c>
Entfernen: <c>
-> Anwendung: Modellierung dynamischer Systeme

Implementierung:
Intuitiv: Liste der Elemente
Vorteil: entfernen in O(1)
Nachteil: Einfügen in O(n) (in Länge der Warteschlange)
Verbesserung: Zusatzzeiger auf letztes Element der Warteschlange
=> Warteschlange = Objekt mit Zeiger auf Anfang + Ende der Elementliste

Beispiel:
Warteschlange mit Elementen a,b,c

Anfang Ende



Realisierung: (define-struct ws (anfang ende))
Konstruktor: (neue, leere Warteschlange):
(define (konstr-warteschlange) (make-ws empty empty)))

Selektoren:
Warteschlange leer?

```
(define (leere-warteschlange? q) (empty? (ws-anfang q)))
```

Anfangselement (ohne Entfernen!)

```
(define (anfang q)
  (if (leere-warteschlange? q)
     (error ’anfang „leere Warteschlange“)
     (first (ws-anfang q))))
```

Mutatoren:
Element hinzufügen:

1. Neues Listenelement konstruieren
2. Element an Ende einfügen (Mutation Ende-Zeiger

```
(define (hinzufuegen-warteschlange! q e)
  (local ((define neues-element (cons e empty)))
    (if (leere-warteschlange? q)
       ´(begin (set-ws-anfang! q neues-element)
               (set-ws-ende! q neues-element)
               q)
        (begin (set-rest! (ws-ende q) neues-element)
              (set-ws-ende! q neues-element)
              q))))
```

Element löschen:
Anfangszeiger auf nächstes Element setzen

```
(define (entfernen-warteschlange! q)
  (if (leere-warteschlange? q)
     (error ….)
     (begin (set-ws-anfang! q (rest (ws-anfang q)))
            q)))
```

Tabellen

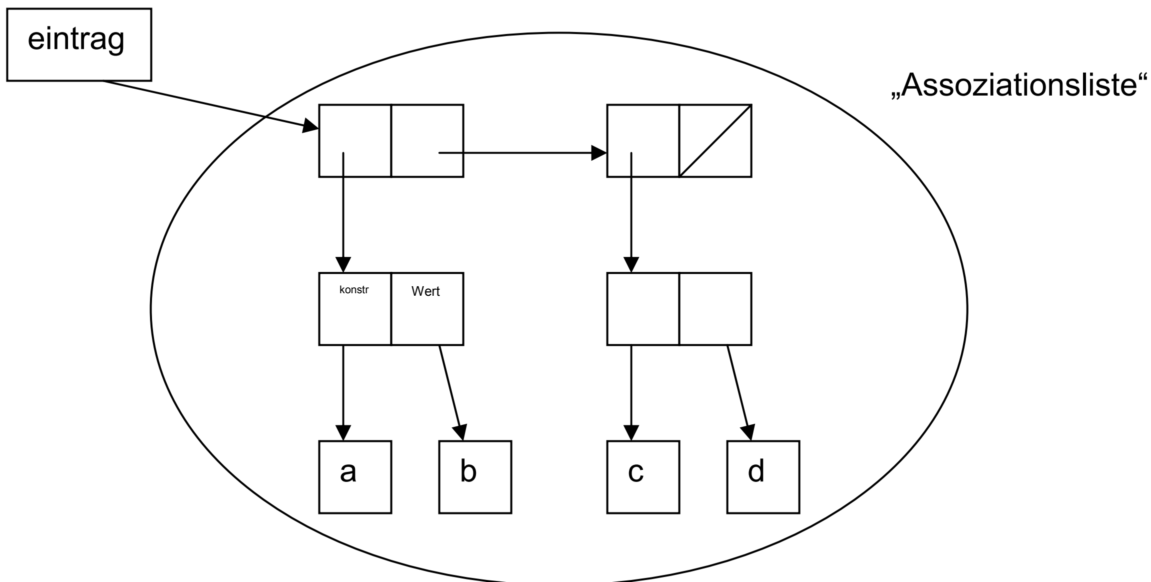
n-dimensional: <key $_1$>   <key $_n$>   ->   Werte

Beispiel (Kapitel 2): Operator-Typ-Tabelle
Operatornamen, Typ -> Prozeduren
Implementierung eindimensionaler Tabellen:

- Liste von Einträgen (key, wert)
  Hierzu:
  (define-struct eintrag (key wert))
  (define-struct tab (einträge))
- Einfügen immer am Anfang der Liste

Beispiel:

Tabelle a -> 1
b -> 2



Konstruktor: leere Tabelle:
(define (konstr-tabelle) (make-tab empty))

Selektor: Suche nach Wert für Selektoren
zunächst: Suche in Assoziationsliste mit Selektor-gleich eq?

```
(define (ass-eq key al)
  (cond ((empty? al) empty)
        ((eq? key (eintrag-key (first al))) (first al))
        (else ass-eq key (rest al)))))
```

Nun:

```
(define (suche-wert key t)
  (local ((define satz (ass-eq key (tab-eintrag t))))
    (if (empty? satz)
        empty
        (eintrag-wert satz))))
```

Mutator: Eintrag neues Schlüssel-Wert-Paar:
Falls Schlüssel vorhanden, dann ändere Wert, sonst hinzufügen

```
(define (eintragen! key wert t)
  (local ((define eintraege (tab-eintraege t))
          (define satz (ass-eq key eintraege)))
  (if (empty? satz)
     (empty? satz)
     (set-tab-eintraege! t (cons (make-eintrag key wert) eintraege)))
     (set-eintrag-wert! satz wert)))))
```

2- und mehrdimensionale Tabellen

Beachte: K1,K2 -> V entspricht K1 -> (K2 -> V)
=> Implementierung einer 2-dimensionalen Tabelle
als 1-dimensionale Tabelle mit Werten als 1-dimensionaler Tabelle
Realisierung aufbauend auf 1-dimensionalem Fall

Tabellen als aktive Objekte

Sichtweise Tabelle ist Objekt, das Nachrichten wie suche-wert und eintragen verarbeitet
=> lokaler Zustand

```
(define (konstr-tabelle)
  (local ((define t (make-tab empty))
          (define (suche-wert key) <wie oben>)
          (define (eintragen! key wert) <wie oben>)
          (define (zuteilen m)
            (cond ((eq? m 'suche-wert-proz) suche-wert)
                  (eq? m 'eintragen-proz) eintrag!)
                  (else (error …))))
  zuteilen))
```

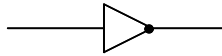
Implementierung einer globalen Tabelle wie in Kapitel 2.3

```
(define op-tabelle (konstr-tabelle))
(define get (op-tabelle 'suche-wert-t-proz))
(define put (op-tabelle 'eintragen-proz))
```

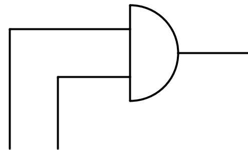
Simulation digitaler Schaltkreise

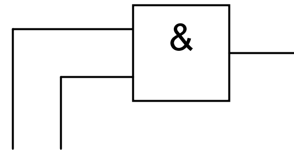
„Computing is simulation“ (Alan Kay)
Hier: Simulation digitaler Schaltkreise
-> Menge von Schaltelementen, verbunden durch Drähte, die den „Wert“ 0 oder 1 haben

Inverter
0 -> 1
1 -> 0

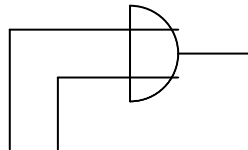


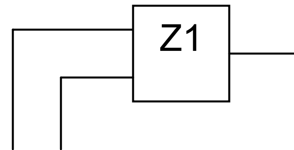
UND-Gatter:
0 0 -> 0
0 1 -> 0
1 0 -> 0
1 1 -> 1





ODER-Gatter:
0 0 -> 0
0 1 -> 1
1 0 -> 1
1 1 -> 1

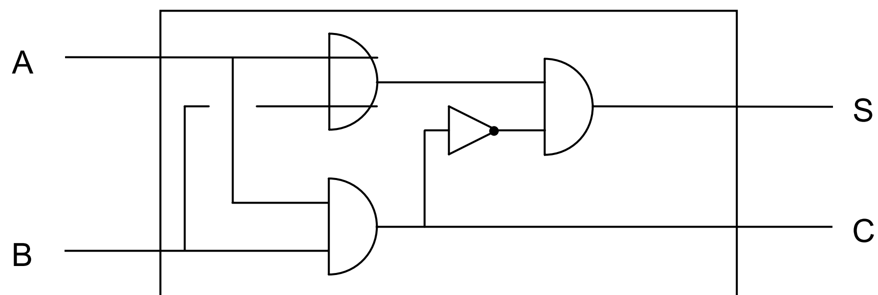




=> Wichtig: Ausgabesignal wird mit Verzögerung aus Eingabesignal erzeugt
=> relevant für Entwurf => Simulation!

Komplexere Schaltungen: Halbaddierer (addiere 2 Bits + Übertrag)

A	B	S	C
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1



Ziel: Simulation der Schaltkreise mit Berücksichtigung der Signalverzögerungen

Objekte in Programm

Drähte
- verbinden Schaltelemente
- haben Wert (0 oder 1)
- veranlassen Aktionen bei Wertänderung

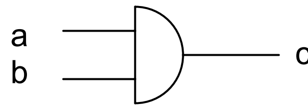
Schaltelemente
- leiten Werte zwischen angeschlossenen Drähten weiter

Beispiel: Drahterzeugung:

```
(define a (konstr-draht))
(define b (konstr-draht))
(define c (konstr-draht))
```

Verbindung mit Schaltelement:

```
(und-gatter a b c)
```



Abstraktion: Definition komplexer Schaltwerke:

```
(define (halbaddierer a b s c)
  (local ((define d (konstr-draht)) (define e (konstr-draht))
    (begin (oder-gatter a b d)
           (und-gatter a b c)
           (inverter c e)
           (und-gatter d e s))))
```

=> Sprache für Schaltkreisentwurf (keine Spezialsprache notwendig!)

Implementierung von „draht“-Objekten:

Konstruktor:	(konstr-draht)	
Selektor:	(set-signal d)	(Wert von Draht d)
Mutator:	(set-signal! d w)	(setze Wert von d auf w)
	(add-vorgang! d p)	p: Prozedur ohne Argumente, wird automatisch bei Wertänderung im Draht d aufgerufen

Globale Prozedur:
(verzoegert zeit proz) Führt „proz“ nach Verzögerung „zeit“ aus

Globale Variablen:
inverter-verzögerung
und-gatter Verzögerungszeiten
oder-gatter

Draht-Implementierung

Lokale Zustände:
signal-wert: Aktueller Wert
vorgang-prozedur bei Wertänderung aufzurufene Prozedur

Modellierung als aktives Objekt:
Noch zu implementieren: verzögert
Realisierung: „Zeitplan“ der aus zuführenden Prozeduren, Operatoren

(leerer-plan? plan) Ist alles abgearbeitet
(erster-plan-eintrag plan) liefert ersten (geordnet über Zeit) Eintrag in plan
(entferne-ersten-plan-eintrag! plan) eintrag löschen
(hinzufügen-plan! zeit voergang plan) Hinzufügen Vorgang in Plan
(aktuelle-zeit plan) Aktuelle Simulationszeit (= Zeit erster Planeintrag)

Globale Variable der-plan:
ein globaler Zeitplan
zur Simulation (fortführen). Führe immer 1. Planeintrag aus, bis Plan leer

Implementierung Zeitplan

Zeitplan ≈ Liste von Zeitsegmenten (Paar (Zeit, Warteschlange)
Warteschlange: Operationen, die zu diesem Zeitpunkt ausgeführt werden müssen
geordnet nach Zeitpunkten / der Zeitgerade

Simulation:

- Definition von Drähten
- Definition von Schaltwerken
- Anbringen von Sonden an Drähten
- Starten mit Signal setzen und (fortfahren)

17.1.2006

Modelle mit Beschränkungen

Traditionelle Programmierung: Berechnung von Werten in eine Richtung
Voraussetzung: Berechnungsrichtung zur Compilezeit bekannt
Alternativ: Richtung unbekannt, Werte durch Relation verknüpft
Ausnutzung der Relation in verschiedene Richtungen sobald Werte bekannt

Beispiel: Temperaturrechnung Celsius – Fahrenheit
Relation: $9C = 5(F - 32)$

- Falls C bekannt: $C = 25 \Rightarrow F = 77$
- Falls F bekannt: $F = 212 \Rightarrow C = 100$

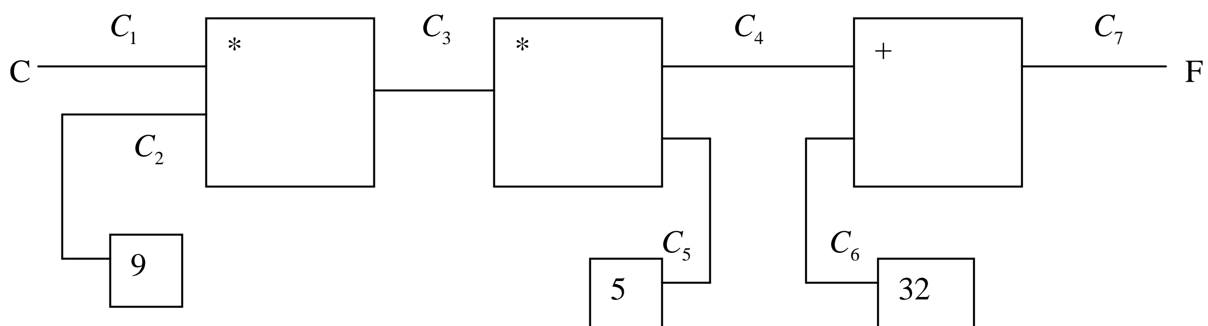
-> Rechnen mit Beschränkungen (Constraint), Propagierung von Werten

Anwendungen:

- Künstliche Intelligenz (Wissensverarbeitung, unvollständige Information)
- Operations Research: Optimierung bezüglich Beschränkung
- Containerverladung
- Produktionsabläufe
- Terminalvergabe bei Flughäfen

Realisierung durch Beschränkungsgesetze bestehen aus

- elementaren Beschränkungen, Relation zwischen Gröen
  (addierer x y z) $\triangleq$ Relation $x + y = z$
  (multiplikator x y z) $\triangleq$ Relation $x \cdot y = z$
  (konstante c x) $\triangleq$ Relation $x = c$
- Konnektoren: Verknüpfungen zwischen Relationen (=> Propagierung von Werten)
  Netzwerk für $9C = 5(F - 32)$



Rechnen im Netzwerk:
Falls Konnektor Wert erhält: Wecke alle angeschlossenen Beschränungen (außer die, von denen der Wert konstant) und informiere diese über neuen Wert
Falls Beschränkung geweckt: Prüfe angeschlossene Konnektoren auf Werte und leite evtl. neue Werte weiter

Beispiel:
Setze C auf Wert 25: $C_1 = 25 \Rightarrow C_3 = 225 \Rightarrow C_4 = 45 \Rightarrow C_7 = 77$
wichtig: umgekehrte Richtung auch möglich!

Implementierung: konstr-konnektor erzeugt neuen Konnektor
(Konnektor: Argument für Beschränkungen)

```
(define C (konstr-konnektor))
(define F (konstr-konnektor))
(define (celsius-fahrenheit-konverter c f)
  (local ((define c2 (konstr-konnektor))
          (define c3 (konstr-konnektor))
          …                              )
    (begin (multiplikator c c2 c3)
           (multiplikator c4 c5 c3)
           (addierer c4 c6 f)
           (konstante 9 c2)
           (konstante 5 c5)
           (konstante 32 c6))))
```

Analogie zur Schaltkreissimulation: Sprache für Beschränkungen

Beobachtung mit Sonden:
(sonde „Celsius“ C)
(sonde „Fahrenheit“ F)

```
>(set-wert! C 25 'benutzer)          benutzer: von wo kommt der Wert?
Sonde: Celsius=25
Sonde: Fahrenzeit=77
```

```
>(set-wert! F 212 'benutzer)
Error: Widerspruch!             <- F hat noch Wert 77
```

daher: Werte löschen

```
>(vergiss-wert! C 'benutzer)
Sonde: Celsius=?
Sonde: Fahrenheit=?
```

```
>(set-wert! F 212 'benutzer)
Sonde: Fahrenheit=212
Sonde: Celsius=100
```

Implementierung: ähnlich wie Schaltkreissimulation, ohne Zeitpläne
Datentyp Konnektor: Operationen

(hat-wert? k)	hat Konnektor K einen Wert?
(get wert k)	liefert aktuellen Wert
(set-wert! k wert informant)	Ein Informant möchte den Wert setzen
(vergiss-wert! k rueckziehender)	Ein „rueckziehender“ möchte Wert vergessen
(verbinde k beschraenkung)	beteilige k an „beschräknung)

Elementare Beschränkungen:

- Objekte, die Nachrichten verarbeiten (-> ich
  -> ich-habe-einen-wert
  -> ich-habe-meinen-wert-verloren

- Sonden: wie Beschränkungen, nur Ausgabe statt Verarbeitung
- Konnektoren: Zustand:
  - wert: Aktueller Wert oder empty
  - informant: Objekt, der Wert gesetzt hat (oder ’benutzer)
  - beschraenkungen: Liste der Beschränkungen, mit denen Konnektor verbindet

18.1.2006

3.4 Zustände in nebenläufigen Systemen

Kapitel 1+2: keine Zustände
=> - einfaches Berechnungsmodell (Substitution)
- Ausdrücke sind „zeitlos"

Kapitel 3: lokale Zustände / veränderbare Datenstrukturen
=> - Umgebungsmodell
- Werte von Ausdrücken sind zeitabhängig

z.B. Bankkonto
```
> (abheben 50)
80
> (abheben 50)
30
```
=> Wert ist abhängig von - Form des Ausdrucks
- Vorgeschichte aller bisherigen Berechnungen

aber: Die reale Welt ist nebenläufig
unabhängige Objekte, die alleine agieren aber sich synchronisieren
gute Modellierung => Anwendungsprogramme auch nebenläufig organisieren
Nebenläufige Anwendungen wichtig:

- Internet
- verteilte Informationssysteme
- Simulation

(Programmiertechniken: herkömmliche Sprachen + prozessorientierte Bibliotheken,
neuere Sprachen mit nebenläufigen Konstanten)

Nebenläufigkeit > Probleme bei Zustandsänderungen

Abheben:
```
(define (abheben betrag)
  (if (>= kontostand betrag)
     (begin (set! kontostand (- kontostand betrag))
            kontostand)
     „Deckung nicht ausreichend“))
```

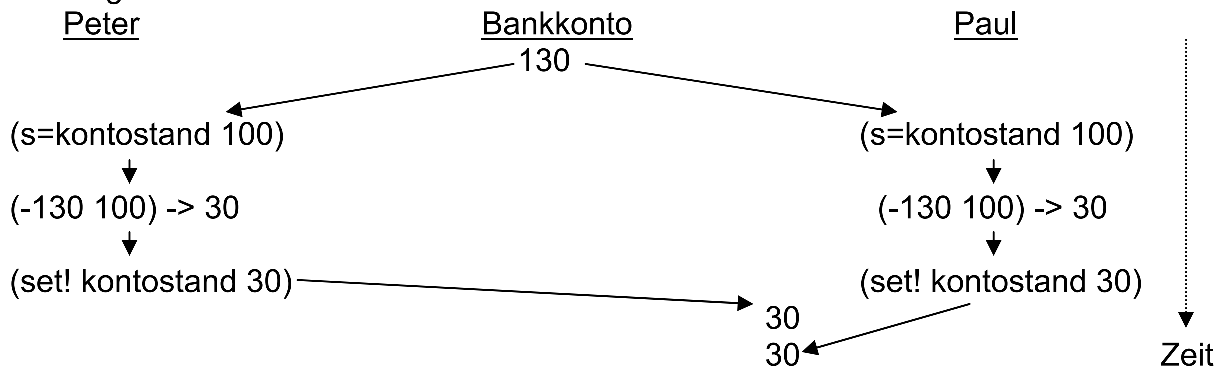
Annahme:

- Peter und Paul haben ein gemeinsames Konto
- Kontostand: 130
- Peter und Paul heben an verschiedenen Geldautomaten 100 ab

Möglichkeiten bei fast gleichzeitiger Abhebung:

1. Beide dürfen abheben
   Peter hebt 100 ab, kontostand: 30
   Paul hebt 100 ab, kontostand: -70
2. beide dürfen abheben
   Peter und Paul heben jeweils 10 ab, kontostand: 30

Zeitdiagramm:



Notwendig: Kontrolle „kritischer“ Programmbereich

<u>Senaphore</u>
Signatur (z.B. Werte 0,1) mit Operatoren P und V
(von „passeren“ / „vrijgeven“, Dijkstra 1968, Signale bei Eisenbahnen)
(P s) falls s=1, setze s auf 0, sonst: stoppe Programmausführung bis s=1
(d.h. Eintrag in Warteschlange)
(V s) setze s=1, falls Prozesse auf s warten, aktiviere einen

wichtig:
P und V unteilbare Operationen
(Betriebssystem garantiert, dass nur ein Prozess P/V ausführen kann)

Verhinderung der Bankpfeile

1. erzeuge Senaphor s
2. (define (konstr-abheben betrag)

```
  (begin (P s)
         (abheben betrag)
         (V s)))
```

=> nur 1 Prozess gleichzeitig abheben

Nachteil: kein gleichzeitiges Abheben von verschiedenen Konten
Lösung:
erzeuge für jedes Konto eigenen lokalen Senaphor

```
(define (konstr-konto kontostand)
  (lokal ((define s (konstr-senaphor))
          (define (abheben betrag))))
```

neue Probleme durch Nebenläufigkeitskontrolle:
<u>Verklemmung</u> (<u>Deadlock</u>)
Beispiel: Überweisung zwischen Konten k1 und k2. Kontrolle mit Senaphoren s1 und s2 für diese Konten

```
(define (ueberweisen k1 k2 s1 s2 betrag)
  (begin (P s1)
         (P s2)
           <Überweisung tätigen>
         (V s2)
         (V s1))
```

Mögliche Situation: Peter und Paul haben verschiedene Konten, wollen gleichzeitig gegenseitig überweisen.
(überweisen peter-k paul-k peter-s paul-s b1)(überweisen paul-k peter-k paul-s peter-s b2)

=> beide warten: Verklemmung!
Vermeidung: sorgfältige Planung an Strategien zu „Verklemmungsvermeidung“
aber: oft schwierig zu überschauende Zeitabhängigkeiten

Fazit:
Zustandsänderung => Zeitaspekte relevant
=> schwierige Kontrolle in realen Systemen
Konsequenzen für Programmentwurf:
- zustandsfrei wo immer es möglich ist
- Zustände lokal halten, alle Änderungen kontrollieren (-> Nachrichtenweitergabe)
- Änderungsoperationen synchronisieren, mögliche Verklemmungen ausschließen

24.1.2006

3.5 Datenströme

- weitere Abstraktionstechnik
- besserer modularer Aufbau
- Alternative zur zustandsorientierten Programmierung

Beispiele

1. Summiere im Binärbaum (Blatt = ganze Zahl) die Quadrate aller ungeraden Blätter

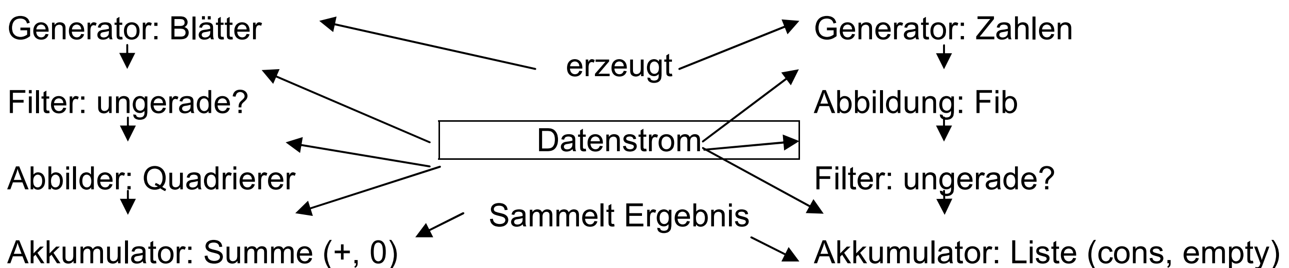
```
(define (sume-uq b)
  (if (blatt? b)
    (if (ungerade? b)
       (quadrat b)
       0)
    (+ (summe-uq (linker-ast b))
       (summe-uq (rechter-ast b)))))
```

2. Liste aller ungeraden Fibonacci-Zahlen Fib(n) mit $1 \le k \le n$

```
(define (ungerade-fibs n)
  (local
    ((define (naechstes k)
      (if (> k n)
         empty
         (local ((define f (fib k)))
           (if (ungerade? f)
             (cons f (naechstes (+ k 1))
                     (naechstes (+ k 1))))))))
    (naechstes 1)))
```

summe-uq	ungerade-fibs
1. Aufzählen aller Blätter	1. Aufzählung ganzer Zahlen
2. Filter die ungeraden	2. Fib-Berechnung
3. Quadrieren	3. Filtern der ungeraden
4. Akkumulieren der Summe	4. Akkumuliere in Liste

Programme verschieden, ähnliche Datenfluss



Die Programme sind verschieden, jedoch besitzen sie einen ähnlichen Datenfluss.

Ziel: Datenflussstruktur im Programm sichtbar machen
Vorteile: übersichtlich, lesbar, universelle Bausteine
Datenströme: Folgen von Elementen, Stromoperationen:

cons-strom: Konstruiere Strom
kopf: 1 Element
rest-strom: restlicher Strom
der-leere-Strom: Strom ohne Elemente
leerer-Strom?: ist Strom leer?

<u>Gesetze:</u>
(kopf (cons-strom a b)) = a (rest-strom (cons-strom a b))
= b (leerer-strom? (cons-strom a b)) = false (leerer-strom? der leere-strom) = true

<u>Implementierung:</u>
z.B. als Listen, später jedoch: spezielle Implementierung

```
(define (append-stroeme s1 s2)
  (if (leerer-strom? s1)
    s2
    (cons-strom (kopf s1)
                (append-stroeme (rest s1) s2))))
```

<u>Strukturierung mit Strömen</u>
Generator: BAUM -> STROM der Blätter

```
(define (gen-baum b)
  (if (blatt? b)
     (cons-strom b der-leere-strom)
     (append-stroeme (gen-baum (linker-ast b))
                     (gen-baum (rechter-ast b)))))
```

<u>Filter:</u> unterade Zahlen:

```
(define (filter-ungerade s)
  (cond ((leerer-strom? s) der-leere-strom)
        ((ungerade? (opf s)) (cons-strom (kopf s) (filter-ungerade (rest-strom s))))
        (else (filter-ungerade (rest-strom s)))))

(define (abb-quadrat s)                          ;Abbildung für Quadrate
  (if (leerer-strom? s)
     der-leere-strom
     (cons-strom (quadrat (kopf s)) (abb-quadrat (rest-strom s))))))

(define (akk-summe s)
  (if (leerer-strom? s) 0
                       (+ (kopf s) (akk-summe (rest-strom s)))))
```

Analog:

```
(define (ungerade-fibs n)
  (akk-cons                ; wie akk-summe, nur: + -> cons, 0 -> empty
    (filter-ungerade
      (a-fib               ; wie abb-quadrat, nur: quadrat -> fib
        (sum-intervall 1 n))))))
```

Nun einfach: neue Operationen zusammensetzen:

Beispiel:
Summe der Quadrate der ersten n Fibonacci-Zahlen

```
(define (summe-qf n)
  (akk-summe
    (abb-quadrat
      (abb-fib
        (gen-intervall 1 n)))))
```

Universelle Prozeduren für Datenströme:
z.B. abb-quadrat / abb-fib zu speziell und ähnlich => universeller Abbilder

```
(define (abb f s)
  (if (leerer-strom? s)
     der-leere-strom
     (cons-strom (f (kopf s)) (abb f (rest-strom s)))))
```

analog: universeller Filter:

```
(define (filter-strom p s)
  (cond ((leerer-strom? s) der-leere-strom)
        ((p (kopf s)) (cons-strom (kopf s) (filter-strom p (rest-strom s)))
        (else (filter-strom p (rest-strom s)))))
```

Universeller Akkumulator:
op: Verknüpfung der Ergebnisse
e: Ergebnis für leerer Strom

```
(define (akk op e s)
  (if leerer-strom? s) e
                      (op (kopf s) (akh op e (rest-strom s)))))
```

=> Neudefinition summe-uq

```
(define (summe-uq b)
  (akk + 0
    (abb quadrat
          (filter-strom ungerade?
                   (gen-baum b)))))
```

Weitere neue Kombinationen: Produkt der Quadrate der ungeraden Zahlen zwischen 1 und n

```
(define (prod-qu n)
  (akk * 1
    (abb quadrat
        (filter-strom ungerade?
              (gen-intervall 1 n)))))
```

Universeller Akkumulator vielfach anwendbar: Strom von Strömen -> einen Strom

```
(define (glaetten s) (akk append-stroeme der-leere-strom s))
```

Stromorientierte Programmierung für viele Datenverarbeitungs-Probleme einsetzbar
gegeben: Strom mit Studierendendaten
gesucht: Liste von Studierenden mit Klausurqualifikation ($\geq$50% Übungspunkte)

```
(define (klausur-qualif stud-strom)
  (akk cons empty
       (filter-strom >=-haelfte-uebungspunkte
                     (filter-strom kocut-informatik-1
                                   stud-strom))))
```

Anwendung einer Prozedur (evtl. Ausgabe / Zustandsänderung) auf jedes Stromelement

```
(define (fuer-jedes proz s)
  (if (leerer-strom? s) 'fertig
                        (begin (proz 8kopf s))
                               (fuer-jedes proz (rest-strom s)))))
```

Ausdrücken eines Stroms: (fuer-jedes display s)

25.1.2006

Implementierung von Datenströmen

Naiv: Strom als Liste: cons-strom -> cons
kopf -> first
rest-strom -> rest

Nachteil: Datenstrukturen evtl. unnötig aufbauen
Stromorientierter Stil:

```
(akk
  (filter-strom …
    (abb
      (abb…
        strom))))
```

Idee: Element eastrom dauerhaft abb, abb, filter-strom, akk
aber: applikative Ordnung => Aufbau des kompletten Eingabestroms in jeder Stufe

Beispiel:
2. Primzahl im Intervall 10000 bis 1000000
(kopf (rest-strom (filter-strom prinzahl? (gen-intervall 10000 1000000))))
=> Aufbau Liste mit fast 1000000 Elementen

Verbesserte Implementierung
Aufbau nur soweit, wie tatsächlich benötigt
cons-strom -> (cons e <Prozedur zur Berechnung des Reststroms>)
kopf -> e
rest-strom -> Aktivierung von <Prozedur zur Berechnung des Reststroms>

Realisierung: <u>verzögerte Auswertung</u>
Scheme-Sonderform:
(delay <Ausdruck>)
keine Auswertung, Ergebnis verzögertes Objekt o, Auswertung mit (force o)
Gesetz:
(force (delay e))=e

Alternativimplementierung:
(cons-strom a b) äquivalent zu (cons a (delay b)) (Sonderform!)
(define (kopf s) (first s))
(define (rest-strom s) (force (rest s)))
Auswertung Reststrom erst dann, wenn benötigt
Es gilt:
(kopf (cons-strom a b))=(kopf (cons a (delay b)))=(first (cons a (delay b)))=a
(rest-strom (cons-strom a b))=(rest-strom (cons a (delay b)))
=(force (rest (cons a (delay b))))=(force (delay b))=b

=> Implementierung auch korrekt!

Effekt der verzögerten Auswertung:
Betrachte:

```
(define (gen-intervall u o)
  (if (> u o)
      der-leere-strom
      (cons-strom u (gen-intervall (+ u 1) o)))
```

Auswertung von
(kopf (rest-strom (filter-strom primzahl? (gen-intervall 10000 1000000))))

1. Anwendung (gen-intervall 10000 1000000)
2. Filter-Anwendung -> (filter-strom primzahl? (rest-strom …))
3. Filter-Anwendung: 10002 keine Primzahl (force „…“) -> (cons-strom 10001)
   … „Verzögerung von
   10007 mit Primzahl (gen-intervall 10002 1000000)
   Filter-Ergebnis:
   (cons 10007 “Verzögerung (filter-strom primzahl? (cons 10008 “Verzögerung…“))“)
4. (rest-strom (cons…)
   -> (filter-strom primzahl? (cons 10008 “Verzögerung (gen-intervall 10000 1000000))
5. 10009 Primzahl: (cons (10000) “Verzögerung (filter-strom …)“)
6. (kopf (cons …)) -> 10009

- kein Komplettaufbau von 10000, 10001, …, 1000000
- nur Stückweise, soweit benötigt
- „aufforderungsgesteuerte“ Anwendung

Hierdurch Kombination von
- datenorientierte Programmstruktur (erzeuge alle Daten, transformiere Daten, akkumuliere)
- inkrementelle Anwendung (erzeuge Element, manipuliere dieses, erzeuge nächstes Element, …)

=> Entkopplung Programmstruktur von Auswertungsstruktur

Anmerkung:
Sprachen mit Auswertung in Normalordnung: keine spezielle Implementierung (d.h. Ströme = Listen

Implementierung der verzögerten Auswertung

Prozeduren:
(define (quadrat x) (* x x))
erst auswerten bei Prozeduranwendung
Idee:
(delay <Ausdruck>) andere Schreibweise für (lambda () <Ausdruck>)
Nun:
(define (force 0) (0))
Nachteil:
verzögerte Objekte duplizieren (mehrfache Vorkommen im Rumpf)
=> dasselbe Objekt mehrfach auswerten

Verbesserung:

- bei Auswertung: speichere Ergebnis
- bei weiterer Auswertung: liefere direkt Ergebnis zurück
- Implementierung: tabelliere Prozedur:

```
(define (tab-proz proz)
  (local ((define bereits-ausgewertet? false)
          (define ergebnis empty))
    (lambda () (if bereits-ausgewertet?
                   ergebnis
                   (begin (set! ergebnis (proz))
                          (set! bereits-ausgewertet? true)
                          ergebnis)))))
```

Dann: (delay <Ausdruck>) ≡ (tab-proz (lambda () <Ausdruck>))

=> Scheme-Implementierung von delay!

<u>Datenströme unendlicher Länge</u>

Beispiel: Strom ganzer Zahlen

```
(define (zahlen-von n) (cons-strom n (zahlen-von (+ n 1))))
(define alle-zahlen (zahlen-von 0))
```

(zahlen-von 0): unendlicher Strom 0,1,2,3,…

Zugriff auf n-tes Stromelement (n=0 ≈ Kopf)

```
(define (n-tes-strom n s)
  (if (= n 0) (kopf s)
          (n-tes strom (- n 1) (rest-strom s 1))))
```

Nun:

```
>(n-tes-strom 5 alle-zahlen)
5
```

Unendlich: Ströme wie üblich bearbeitbar

Beispiel:

Ströme wie üblich bearbeiten

Strom aller Zahlen, die nicht durch 7 teilbar sind

```
(define (teilbar? x y) (= (remainder x y) 0 )
(define ohne-sieben (filter (lambda (x) (not (teilbar? x 7))) alle-zahlen))
```

Nun:

```
>(n-tes-strom 50 ohne-sieben)
57
```

Beispiel: Strom aller Fibonacci-Zahlen

```
(define a b) (cons-strom a (fibgen b (+ a b))))
(define (fibgen 0 1))
```

- rekursive Definition (ohne iterative)
- lineare Laufzeit!

31.1.2006

Unendliche Ströme: Primzahlen

Sieb des Eratosthenes

Idee:
- konstruiere ganze Zahlen ab 2
- steiche im Reststrom alle durch 2 teilbaren Zahlen
- Ergebnis: Strom beginnend mit 3
- streiche in diesem Reststrom alle durch 3 teilbaren Zahlen

Generell:
- streiche im Rest eines Stromes (beginnend mit x) alle durch x teilbaren Zahlen

Ergebnis:
- Köpfe aller konstruierten Teilströme sind Primzahlen

Implementierung:
```
(define (siebe strom)
  (cons-strom (kopf strom)
    (siebe (filter-strom
      (lambda(x) (not (teilbar? x (kopf-strom)))
                   (rest-strom strom)))))
(define primzahlen (siebe (zahlen-raum 2)))
```

```
>(n-tes-strom 50 primzahlen)
233
```

Datenströme <-> lokale Zustände

Was sind lokale Zustände?
Größen, die sich zeitlich ändern können

Wert	0	6	8	5	7
Zeit	t_0	t_1	t_2	t_3	t_4

Vorteile:
- Vermeidung von set!
- Vermeidung der Probleme von Zustandsänderung
- durch verzögerte Auswertung nur Teile des Wertstroms erzeugt

Beispiel: Bankkonto
```
>(define k (konstr-abheben 200))
>(k 30)
170
>(k 40)
130
```

Alternative:
Bankkonto:
Eingabe: Kontostand + Strom von Abhebungen
Ausgabe: Strom von Kontoständen

```
(define (strom-abheben kontostand betrag-strom)
  (cons-strom kontostand
    (strom-abheben (- kontostand (kopf betrag-strom))
                   (rest-strom betrag-strom))))
```

strom-abheben: wohldefinierte mathematische Funktion (keine Seiteneffekte)

Beispiel: Monte-Carlo-Simulation (vgl. Kap. 3.1)

```
(define zufall
  (local ((define x zufall-init))
    (lambda () (begin (set! x (zufall-aktuell x)) x))))
```

alternativ:

```
(define zufallszahlen
  (cons-strom zufall-init (abb zufall-aktuell zufallszahlen)))
```

(implizierte Stromdefinition!)

nun: Cesaro-Test auf Zufallszahlen-strom:

- Abbilder für binäre Abbildung auf aufeinander folgende Zahlen:
  ```
  (define (abb-paare f s)
    (cons-strom (f (kopf s) (kopf (rest-strom s)))
      (abb-paare f (rest-strom (rest-strom s)))))
  ```
- Strom der Ergebnisse des Cesaro-Tests:
  ```
  (define cesaro-strom
    (abb-paare (lambda (z1 z2)
      (= (ggT z1 z2) 1))
      zufallszahlen))
  ```

Ergebnis-strom: Auswertung á la Monte Carlo

```
(define (monte-carlo exp aw af)
  (local
    ((define naechstes aw af)
      (cons-strom (/ aw (aw af))
      (monte-carlo (rest-strom exp) aw af))
      (if (kopf exp)
        (naechstes (+ aw 1) af)
        (naechstes aw (+ af 1))))

(define pi-strom
  (abb (lambda (p) (wurzel (/ b p)))
       (monte-carlo cesaro-strom 0 0)))
```

Struktur:

Strom von Zufallszahlen -> Cesaro-Test -> Strom von Experimentenumgebung
-> Monte-Carlo-Auswertung -> Strom von Approximation

Diskussion: Datenströme <-> Objekte mit Zuständen
- Zeitliche Änderung als Ströme modellieren
- Verwendung der theoretischen Probleme bei Zustandsänderung

Funktionale Programmiersprache
- Prozeduren = (math.) Funktionen ihrer Argmente
- keine Zuweisung
- „scheme ohne set!“

Vorteile:
- einfache Korrektheitsbeweise
- hochgradig für Parallelverarbeitung
  a) parallele Auswertung von Teilstücken
  b) keine Synchronisationsprobleme bei Zuweisung der gemeinsamen Variablen
     => wichtiges Berechnungsmodell für nebenläufige Systeme (Beispiel: Erlang)

Offenes Problem: Zuweisung immer vermeidbar?

schwierig: Modellierung von interaktiven Systemen:
(natürlicher) Ansatz: Objekte mit lokalem Zustand, die Nachichten austauschen mit
Strömen: Sequenzialisierung der Nachrichten als Strom
Beispiel: Peter und Paul teilen sich ein Bankkonto.

- Objektorientiert: Bankkonto = Objekt, Zustand „betrag“ reagiert auf Nachrichten von Peter und Paul
- Stromorientiert: Wir liefern Peter und Paul Nachrichten an Eingabestrom?
  Lösung: Mischen von Nachrichten

<u>Fairer / nichtdeterministisches Mischen</u>

- nimmt aus einer der beiden Ströme Eingabe, falls vorhanden
- wechselt „fair“ zwischen den Strömen (keiner muss beliebig lange wachsen)

=> Zeitbegriff wird relevant: „nicht deterministische Funktion“
=> Erweiterung funktionaler Sprachen notwendig
weiterer Nachteil stromorientierter Modelle:
- Aufteilung in Eingabe und Ausgabe (ströme)
- problematisch: falls Aufteilung unklar, z.B. bei Beschränkungsnetzen
  A+B=C ist Relation, Eingabekonten A und B, aber auch B und C sein
  => relative statt funktionale Sichtweise

1.2.2006

Logische Programmiersprachen (z.B. Prolog)

- elementare Einheiten: Relationen statt Funktionen
  Beispiel: Relation istMutterVon. Aussagen: Helga istMutterVon Sabine, Christine istMutterVon Helga
- Rechnen mit Schlussfolgerungen: Implikationen (statt Prozedurrümpfen)
  Beispiel: A istMutterVon B und B istMutterVon C => A istGroßmutterVon C
- keine festgelegten Ein- / Ausgaben. X istGroßmutterVon Sabine? Antwort: X=Christine
- Chirstine istGrossmutterVon Y? Antwort: Sabine

Konsequenz: logisches Programm zum symbolischen Differenzieren => Integralberechnung mittels unbekannter Eingabe

Zusammenfassung

- kein universelles Programmierparodigma, sondern imperativ / zustandsorientiert, funktional, datenorientiert, nebenläufig, relational
- keine universelle (d.h. für jedes Problem gleich gut geeignet) Programmiersprache
- wähle je nach Problem geeignete Modellierung und Sprache
- Tendenz:
  a) große interaktive Systeme: zustands-/objektorientiert
  b) algorithmische Aspekte (Korrektheit!): datenorientiert / funktional
  c) Datenbanken: relational
  d) Integration?

4. Einführung in Java

4.1 Allgemeines

Scheme: einfach erlernbar, flexibel (unterschiedlche Programmstile), kompatible Programme

Entwicklung großer Systeme:

- gleiche Programmiertechniken (Abstraktion)
- mehr Programmiersicherheit -> Deklarationen von Programmobjekten (=> Prüfung durch Compiler) => größere Programme mit Redundanzen

Java:

- imperative, objektorientierte Sprache
- Typdeklarationen
- Module (packages)
- Prozesse
- plattformunabhängig
- Sicherheitsprüfungen möglich
- automatische Speicherverwaltung

Grundsätzliches Arbeiten mit Java:
edit -> javac -> java (edtieren -> übersetzen -> ausführen)

4.2 Ausdrücke + Anweisungen
Kommentare:
// bis zum Zeilenende
/* über mehrere Zeilen */

Namen:
- beginnen mit Buchstaben, gefolgt von Buchstaben, Ziffern, -, „$"
- Groß-/Kleinschrift relevant, d.h. mod≠MOD

Java: streng getypte Sprache!
jede Variable hat Typ (=Wertmenge) und wird vom Compliler geprüft

Scheme:	Java:
(define x 5)	int x=5;
(set! x true)	x=true;
	-> Error, incompatible Typen – auch x=3,14;

Grundtypen in Java

Typname	Werte	Initialwert
Boolean	True, false	False
Char	16-bit-Unicode Zeichen 'a' '\u' 'u1111'	\u0000 Unicode: weltweiter Zeichensatz
Byte	8-bit ganze Zahl mit Vorzeichen	0 -128…127
Short	16-bit ganze Zahl mit Vorzeichen	
Int	32-bit ganze Zahl mit Vorzeichen	
Long	64-bit ganze Zahl mit Vorzeichen	
Float	32-bit-Gleitkommazahl	0.0 312.159E-2f, sonst double
Double	64-bit-Gleitkommazahl	
String	Zeichenketten	„Hello"

Ausdrücke		Scheme
Konstanten:	True, false, 11	…
Variablen:	X,y	X y
Aufrufe	f($a_1,..,a_n$)	(f $a_1,..,a_n$)
Operatoren:	a_1 op a_2	(op a_1 a_2)

Problem: Mehrdeutigkeiten: 1-2-3 -> (- 1 (- 2 3))
-> (- (- 1 2) 3)
Lösung: Festlegung von Präzedenzen, Prioritäten, Assoziativitäten

Anweisungen:
Java: Variablen: Namen für veränderbare Speicherzellen relevant. Veränderung der Variablenwerte durch Zuweisungen

Zuweisung:

Java	Scheme
x=a; (Austausch / typkompatibel)	(set! x a)
Kurzformen:	
x++;	(set! x (+ x 1))
x+=5;	(set! x (+ x 5))

Anweisungsfolgen:

Java	Scheme
$\{a_1,..,a_n\}$	(begin a_1 ... a_n)

Prozeduraufruf:

Java	Scheme
p($a_1,..,a_n$);	(p $a_1,..,a_n$)

p: Funktion ohne Rückgabewert

Fallunterscheidung:

Java	Scheme
if (ausdruck) anweisung	(if ausdruck anweisung 'fertig)
if (a) anw1 else anw2	(if a anw1 anw2)
switch (ausdruck){	(local ((define x ausdruck))
case k_1 : a_1; break;	(cond ((eq? x k_1) a_1)
case k_n : a_n; break;	((eq? x k_n) a_n)
default: a_{n+1}	(else a_{n+1})))
}	

Schleifen:
statt Rekursion in Scheme:

while (ausdruck) anw

```
(define (while b x)
  (cond ((b) (begin (x) (while b x))))
(while ((lambda () ausdruck (lambda () anw))
```

do anw while (ausdruck)

```
(define (repeat x b))
  (begin (x)
         (cond ((b) (repeat x b))))
(repeat ((lambda () anw) (lambda() ausdruck))
```

for (init; bexp; iexp) anw — {init; while(bexp) | {anw; iexp;}}

init: Initialisierung
bexp: Abbruchbedingung
iexp: Inkrementierung

Beispiel:

```
for (int i=1; i<=n; i++)
  {x=x+2+i;}
```

Sprünge:
eingeschränkt, „strukturierte" Sprünge

break;	Beende Schleife / switch, in dem sich dieses break befindet
continue;	Verlasse Schleifenrumpf, nächste Iteration
return;	Verlasse Prozedur
return <Ausdruck>;	Verlasse Funktion mit Rückgabewert <Ausdruck>

Ausnahmebehandlung:

Fehlersituation kontrollieren (statt Absturz!)
Konzept:
Anweisungen können Fehler (exceptions) auslösen
Anweisung zur Behandlung von Fehlern

```
try {<normale Berechnung>}
catch (ExceptionType) {Behandlung des Fehlers}
…
catch (ExceptionType) {Behandlung des Fehlers}
finally {<immer auszuführender Code>}
```

7.2.2006

4.3 Deklarationen

Prinzip:
Alle im Programm verwendeten Namen müssen deklariert werden (falls nicht vordefiniert).

Beispiel (Scheme):
```
>(define l (cons 1 empty))
>(+ l 1)        -> Error. Illegal type…
```

Dies könnte vom Compiler erkannt werden, da l vom Typ Liste

=> in Java:
Jedes Objekt hat einen festen Typ, der bei der Deklaration angegeben werden muss.
Typ ≈ Menge zulässiger Werte.

Variablendeklaration (als Anweisung)
<typ> <name>;
<typ> <name>=<ausdruck>; => Initialwert
auch mehrere Namen: i,j,k;

Konstantendeklarationen
wie Variablendeklaration, aber mit Präfix „final" nur eine Zuweisung erlaubt
final double pi=3,1415926

Anforderung an Zuweisung: x=yi
Typ von y muss in Typ von x (verlustfrei) konvertierbar sein, d.h. Wert von y ist auch zulässiger Wert für Typ von x
z.B.
```
double x = 1.5;
int i = 12;

x=i;          //zulässig
i=x;          //unzulässig
i=(int) x     //zulässig
```
-> „type cast": erzwungene Typkonversion, evtl. mit Informationsverlust
```
i=true;       //unzulässig
```

Beachte: wegen Typanforderung manche Programmiertechniken nicht anwendbar, z.B. Nachrichtenweitergabe,
dafür Ersatz in Java: Klassendefinitionen

Felder (engl. arrays)
geordnete Menge von indizierten Variablen, d.h. Abbildung [0..n]->Variablen
ähnlich: Tabellen, nur fester Indexbereich und keine Erweiterung

Deklaration: int[]a;
int: Typ der Feldelemente
[ ]: Größe bei Erzeugung festlegen

Erzeugung:

- dynamische Generierung a=new int[100];
  Größe / Anzahl der Feldelemente
  Indexbereich: 0..99
- Initialisierung: a={1,2,4,8,16,32};
  Feld mit Indexbereich 0..5

Zugriffe auf Elemente: Indizierung: a[i]≈Feldelement mit Index i

Beispiel: Aufsummieren aller Feldelemente

```
int[ ] a= {1,2,4,8,16};
int s=0;
for (int i=0; i<5; i++) 5+=a[i];
```

auch mehrdimensionale Felder sind möglich: int[ ][ ]n=new int[5][10];

<u>Funktionen, Prozeduren</u>

Java objektorientiert => Prozeduren <u>neu</u> in Kombination mit Objekten / Klassen: <u>Methoden</u>

Deklaration enthält:

- Funktionsname
- Ergebnistyp
- Parameternamen und –typen
- Prozedurrumpf

allgemein: <typ> <name> (<typ1> <parameter1>, …, <typn> <parametern>) {<Rumpf>}

typ: Ergebnistyp

≈Scheme: (define (<name> <parameter1> … <parametern>) <Rumpf>

Beispiel:

```
int quadrat (int x) {return x*x;}
int x=quadrat(4);
```

Aufruf: aktuelle Parameter(werte) müssen wie bei Zuweisung typkonvertierbar in formal deklarierte Parameter sein

Ergebnistyp: <u>void</u>

- liefert kein Ergebnis
- Aufruf ≈ Zuweisung
- return ohne Ausdruck

```
void display(int x) {System out.println(x);}
display(99);
```

4.4 Klassen und Objekte

Java-Programm ≈ Definitionen von Klassen, die das Verhalten von Objekten beschreiben
-> Ausführung: sende Nachricht „main“ an bestimmte Klasse

Objekte

- haben lokalen Zustand (Attribute)
- verstehen Nachrichten
- Struktur und Verhalten beschrieben in Klassen

Merkmal eines Objekts: Attribute + Nachrichten

Klassen

- beschreiben Eigenschaften von Objekten
- Variablendeklarationen (Attribute, lokaler Zustand)
- Methodendeklarationen (Implementierung der Nachrichten)
- Konstruktordeklarationen (Erzeugung von Objekten)

generelle Struktur:

```
class C {
  <Variablendeklarationen>
  C(…) {<Initialisierung>}            //Konstruktoren
  <Methodendeklarationen>
}
```

Bedeutung: wie Nachrichtenweitergabe in Scheme, aber Definition von „zuteilen“ nicht nötig

- Konstruktoren:
  Prozeduren mit gleichen Namen wie Klasse, werden bei Objekterzeugung aufgerufen
  auch mehrere mit verschiedenen Parametern
- Objekterzeugung:
  new var Klassenname + Argumente für Konstruktor
  d.h. new C(…) -> Referenz auf neues Objekt (≈Prozedur „zuteilen“)
- Variable von Objekttyp: C x;
- Nachricht m senden an Objekt o: Scheme ((o ’m) p1 … pn)
  in Java: o.m(p1,…,pn)
  Sichtbarkeiten von Merkmalen: optionales Sichtelement von Merkmalen:
  public: überall sichtbar
  private: nur in „Klasse“ selbst sichtbar
  protected: nur in Klasse, Unterklasse (später) und in gleichen Paket sichtbar
  keine Angabe: nur innerhalb der Klasse und dem gleichen Paket sichtbar
  z.B. könnte „main“ neues public sein!
- Statische Merkmale:
  „static“ vor Merkmalen => Merkmal gehört zur Klasse und nicht zu Objekten der Klasse
  a) nur einmal repräsentiert (auch ohne Existenz von Objekten)
  b) Zugriff von außerhalb <Klassenname>.m…

Beispiel: Durchnummerierung aller Bankkonten

```
class Konto{
  int kontostand,kontonr;
  private static int (fdnr=0;
  Konto(int i) {
    kontostand=i
    kontonr=fdnr;
    fdnr++;
}
```

besser ist kontonr;

Num: public static void main (String[ ] args) …
public: Zugreifbar von außen
static: zur Klasse
void: kein Ergebnis
args: Aufrufparameter